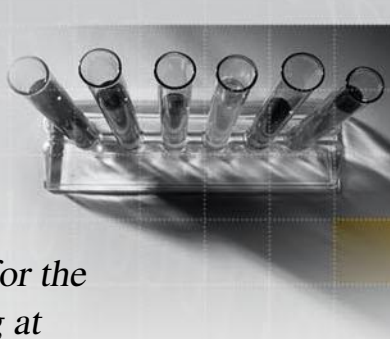
Parallel paradigms in optimal structural design

by

Salomon Stephanus van Huyssteen



Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Mechanical Engineering at Stellenbosch University

Department of Mechanical and Mechatronic Engineering,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Supervisor: Prof. A.A. Groenwold

December 2011

FAKULTEIT INGENIEURSWESE
FACULTY OF ENGINEERING

UNIVERSITEIT
STELLENBOSCH
UNIVERSITY

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the owner of the copyright thereof (unless to the extent explicitly otherwise stated) and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: . . . . . .

Abstract

Parallel paradigms in optimal structural design

S.S. van Huyssteen

Department of Mechanical and Mechatronic Engineering,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Thesis: MScEng (Mech)

December 2011

Modern-day processors are not getting any faster. Due to the power consumption limit of frequency scaling, parallel processing is increasingly being used to decrease computation time. In this thesis, several parallel paradigms are used to improve the performance of commonly serial SAO programs. Four novelties are discussed:

First, replacing double precision solvers with single precision solvers. This is attempted in order to take advantage of the anticipated factor 2 speed increase that single precision computations have over that of double precision computations. However, single precision routines present unpredictable performance characteristics and struggle to converge to required accuracies, which is unfavourable for optimization solvers.

Second, QP and dual are statements pitted against one another in a parallel environment. This is done because it is not always easy to see which is best a priori. Therefore both are started in parallel and the competing threads are cancelled as soon as one returns a valid point. Parallel QP vs. dual statements prove to be very attractive, converging within the minimum number of outer iterations. The most appropriate solver is selected as the problem properties change during the iteration steps. Thread cancellation poses problems caused by threads having to wait to arrive at appropriate checkpoints, thus suffering from unnecessarily long wait times because of struggling competing routines.

Third, multiple global searches are started in parallel on a shared memory system. Problems see a speed increase of nearly 4× for all problems. Dynamically scheduled threads alleviate the need for set thread amounts, as in message passing implementations.

Lastly, the replacement of existing matrix-vector multiplication routines with optimized BLAS routines, especially BLAS routines targeted at GPGPU technologies (graphics processing units), proves to be superior when solving large matrix-vector products in an iterative environment.

These problems scale well within the hardware capabilities and speedups of up to 36× are recorded.

Uittreksel

Parallele paradigmas in optimale strukturele ontwerp

("Parallel paradigms in optimal structural design")

S.S. van Huyssteen

Departement Meganiese en Megatroniese Ingenieurswese,
Universiteit van Stellenbosch,
Privaatsak X1, Matieland 7602, Suid-Afrika.

Tesis: MScIng (Meg)

Desember 2011

Hedendaagse verwerkers word nie vinniger nie as gevolg van kragverbruikingslimiet soos die verwerkerfrekwensie op-skaal. Parallelle prosesseering word dus meer dikwels gebruik om berekeningstyd te laat daal. Verskeie parallelle paradigmas word gebruik om die prestasie van algemeen sekwensiële optimeringsprogramme te verbeter. Vier ontwikkelinge word bespreek:

Eerste, is die vervanging van dubbel presisie roetines met enkel presisie roetines. Dit poog om voordeel te trek uit die faktor 2 spoed verbetering wat enkele presisie berekeninge het oor dubbel presisie berekeninge. Enkele presisie roetines is onvoorspelbaar en sukkel in meeste gevalle om die korrekte akkuraatheid te vind.

Tweedens word QP teen duale algoritmes in 'n parallel omgewing gebruik. Omdat dit nie altyd voor die tyd maklik is om te sien watter een die beste gaan presteer nie, word almal in parallel begin en die mededingers word dan gekanselleer sodra een terugkeer met 'n geldige KKT punt. Parallele QP teen duale algoritmes blyk om baie aantreklik te wees. Konvergensie gebeur in alle gevalle binne die minimum aantal iterasies. Die mees geskikte algoritme word op elke iterasie gebruik soos die probleem eienskappe verander gedurende die iterasie stappe. "Thread" kanseleering hou probleme in en word veroorsaak deur "threads" wat moet wag om die kontrolepunte te bereik, dus ly die beste roetines onnodig as gevolg van mededinger roetines was sukkel.

Derdens, verskeie globale optimerings word in parallel op 'n "shared memory" stelsel begin. Probleme bekom 'n spoed verhoging van byna vier maal vir alle probleme. Dinamiese geskeduleerde "threads" verlig die behoefte aan voorafbepaalde "threads" soos gebruik word in die "message passing" implementerings.

Laastens is die vervanging van die bestaande matriks-vektor vermenigvuldiging roetines met geoptimeerde BLAS roetines, veral BLAS roetines wat gerig is op GPGPU tegnologië. Die GPU

roetines bewys om superieur te wees wanneer die oplossing van groot matrix-vektor produkte in ’n iteratiewe omgewing gebruik word. Hierdie probleme skaal ook goed binne die hardeware se vermoëns, vir die grootste probleme wat getoets word, word ’n versnelling van 36 maal bereik.

Acknowledgements

I would like to acknowledge all the people who helped contribute to my project. First of all I want to thank my father, for his encouragement and support during the difficult times, as well as my mother and my two siblings. Secondly, I would like to express my sincere gratitude to my supervisor Prof. Groenwold, who always had the patience to find bugs in my code and guide me through the editing process. I would also like to acknowledge the following people:

- Mick Pont of The Numerical Algorithms Group (NAG), who generated a single precision `E04NQF`library, `lib_e04nqf_single`.
- Mat Cross, also from NAG, who assisted with debugging support.
- Evan Lezar, who helped iron out some bugs during programming of the GPU implementations of `gemv`.
- Lastly, Eric Bainville, who supplied source code for OpenCL-gemv matrix-vector product kernels. Please see his web page, `www.bealto.com`.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Sequential approximate optimization (SAO) methods are the methodology of choice for simulation-based optimization, in particular when the simulations are computationally demanding. Examples include simulations that require the solution of large finite element (FE) or computational fluid dynamics (CFD) models. This thesis introduces recent advances in mathematical optimization and how they are being applied in modern-day processors. The introduction describes the essentials of optimization and parallel processing. When these two technologies are combined it is possible to increase the performance of optimization programs. With the introduction of some novel parallel algorithms into the SAO paradigm, this thesis aims to improve the efficiency of SAO routines.

1.1 Optimization

Search in optimization refers to the choosing of the best element(s) from a set of available alternatives. This is normally done in the process of minimizing (or maximizing) some real function f, called the *objective function*. In general, the the objective function has to be minimized or maximized with respect to the variable(s) and is normally subject to certain limitations, also known as the *constraints*. The latter can be inequality, equality and side constraint functions, or any combination of the above (when no constraints are present the problem is unconstrained).

A general nonlinear constrained optimization problem may be written as:

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathcal{X}\subset\mathcal{R}^n} \; & f_0(\boldsymbol{x}), && \boldsymbol{x} = [x_1, x_2, ..., x_n]^T, \\ \text{subject to} \; & g_j(\boldsymbol{x}) \leq 0, && j = 1, 2, \ldots, m, \\ & h_k(\boldsymbol{x}) = 0, && k = 1, 2, \ldots, r, \\ & \check{x}_i \leq x_i \leq \hat{x}_i, && i = 1, 2, \ldots, n. \end{aligned} \tag{1.1.1}$$

$f_0(\boldsymbol{x})$, $g_j(\boldsymbol{x})$ and $h_k(\boldsymbol{x})$ are scalar functions of the real column vector of design variables $\boldsymbol{x}$. Above, $g_j(\boldsymbol{x})$ represents the inequality constraints and $h_k(\boldsymbol{x})$ represents all equality constraints. The side constraint equations are represented by the last equation and define the upper ($\hat{x}_i$) and lower ($\check{x}_i$) bounds on the primal variable(s) $\boldsymbol{x}$ (Vanderplaats, 2001).

Sequential approximate optimization (SAO) forms the backbone of this study and is used throughout to solve nonlinear constrained optimization problems. SAO constructs successive approximate analytical subproblems $\tilde{\mathcal{P}}[k]$, $k = 1, 2, 3, \ldots$ at successive approximations $\boldsymbol{x}^{\{k\}}$ to the solution $\boldsymbol{x}^*$. Routine-related modules that represent numerical routines and/or solvers are used in the SAO search steps. Some of these functions may be replaced with similar, parallel routines to give identical output.

1.2 Parallelism

Until recently, very little research had been done regarding the advantages parallel algorithms may have on SAO programs. Even though "sequential" refers to the serial nature in which SAO steps are performed, there are several instructions that ensemble the SAO program, instructions that could benefit by being parallelized. Currently, most optimization programs are constructed of serial streams of instructions. Only one instruction may execute at a time, and only once that instruction has been completed may the next be started. This program is therefore generally executed on an uniprocessor architecture or on a single core of a multi-core processor. This means that, like most other programs, general SAO programs are *computation bounded*, and an increase in processing frequency would relate to a decrease in runtime (Hennessy and Patterson, 2003). Unfortunately, increasing the frequency in turn increases the amount of power consumed by a processor according to the power consumption formulation, $P = CV^2 f$, in one compute unit. Recently, the amount of power that can be dissipated by a single processor reached its limit, where P is the power consumed, C is the capacitance being switched per clock cycle[1], V is the voltage and f is the processor frequency measured in cycles per second (Rabaey, 1996).

In 2004, the release of the Tejas and Jayhawk[2] was delayed and finally cancelled on May 7. The eventual cancellation was attributed to these heat dissipation problems, due to extreme power consumption by this single core, with a slated operating frequency of 7 GHz (Tweakers.net, 2007). The cancellation of the Tejas and Jayhawk reflected Intel's intention to focus on dual-core chips and also marked the end of frequency scaling as the dominant computer architecture paradigm (Laurie, 2004).

To enhance computational power, the only alternative option is to add additional hardware in parallel; the additional transistors are now no longer used for frequency scaling. In effect, this transition enabled the continuation of Moore's Law (Moore, 1965) to this day by circumventing power consumption issues. Formerly strictly sequential programs such, as SAO routines, can now be parallelized.

However, the speed increase of parallel algorithms is governed by Amdahl's Law (Amdahl, 1967). Amdahl's Law states that the expected speedup of all code that can run in parallel is equal to the number of processors. For example, if only 80% of the code can be run in parallel, the maximum speed increase one can expect on 4 processors is 2.5, on 16 it is 4 and on 32 it

[1]The capacitance switched per clock cycle is proportional to the number of transistors whose inputs change (Rabaey, 1996).

[2]Tejas was Intel's code name for the successor to the Pentium 4 Prescott core and Jayhawk was the code name for the Xeon counterpart (Laurie, 2004).

is only 4.4. Some of the limitations that impair linear speedups are overheads during thread creation and cancellation, thread synchronization and memory access.

If α is the parallel fraction of the code and c is the number of processors, then

$$S = \frac{1}{\alpha/c + (1-\alpha)} \tag{1.2.1}$$

is the maximum speedup that may be attained by parallelizing the program. This then puts an upper limit on the usefulness of adding more parallel execution units.

One advantage that Amdahl does not account for is the assumption of a fixed problem size, and that the runtime of the sequential section of the program, is independent of the number of processors. This allows parallel applications to scale better than expected when the computation outweighs overheads.

1.3 Merging technologies

Given here are three snippets of a serial SAO flat profile run. A serial SAO routine is applied to two convex problems. One is an unimodal structural optimization problem, and the other is a global optimization problem. These problems are described in depth in Sections 4.4.4 and 6.4.2 respectively. These snippets show some caveats where SAO displays the potential to be parallelized.

The first profile run shows the effort going into the solution of the subproblems when using a dual solver for the n-variate cantilever beam problem defined in Section 4.4.4; the problem is of size $n = 2000$.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
47.73	14.58	14.58	11	1.33	1.90	drive_galqp_
15.45	19.30	4.72	11	0.43	0.43	sparse_by_rows_a_
9.95	22.34	3.04	12	0.25	0.25	strv_
5.27	23.95	1.61	11	0.15	0.58	qp_pre_
4.78	25.41	1.46	12	0.12	0.12	dgemv_
3.57	26.50	1.09	12	0.09	0.09	dgerx1_
3.37	27.53	1.03	12	0.09	0.09	saoi_grads_
3.21	28.51	0.98	11	0.09	0.09	sparser_

Notice how approximately 50% of the runtime is spent solving the subproblems, which would relate to a theoretical increase of two times. Two methods are tested to try to reduce the effort going into the solution of these subproblems.

The second profile shows the behaviour of a global optimization problem. Global optimization problems are embarrassingly parallel by nature.

```
index % time    self  children    called     name
                0.00   10.89      18/18          saoi_splita_ [2]
[1]     99.9    0.00   10.89      18          sao_dense_ [1]
                0.04   10.85     212/212         drive_lbfgsbf_ [6]
                0.00    0.00     183/183         form_kkt_ [36]
                0.00    0.00     230/230         store_iter_ [37]
                0.00    0.00    1250/1252        saoi_seconds_ [41]
                0.00    0.00     395/395         saoistatus_ [42]
                0.00    0.00     377/377         norm2b_ [43]
                0.00    0.00     377/377         norm2f_ [44]
                0.00    0.00     230/230         saoi_funcs_ [47]
                0.00    0.00     212/212         sparser_ [52]
                0.00    0.00     183/183         formgrad_ [57]
                0.00    0.00     183/183         form_act_ [56]
                0.00    0.00     183/183         diahess_ [54]
                0.00    0.00     183/183         strv_ [61]
                0.00    0.00     183/183         saoi_special_ [60]
                0.00    0.00     183/183         fx_kkt_ [58]
                0.00    0.00     165/370         ranmar_ [45]
                0.00    0.00      66/66          conserve_ [62]
                0.00    0.00      18/230         dpmeps_ [46]
```

In this example, the search trajectory requires 18 iterations to find the minimum to a given probability. All 18 of these searches may be performed on separate processors.

The third profile shows the effort going into solving dense matrix vector multiplications. This time the problem is of size $n = 500$.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 93.24     17.37    17.37    10032     0.00     0.00  dgemv_
  1.23     17.60     0.23     2261     0.00     0.00  subsm_
  1.13     17.81     0.21     2274     0.00     0.00  cauchy_
  0.91     17.98     0.17     2261     0.00     0.00  formk_
  0.70     18.11     0.13   502240     0.00     0.00  ddot_
  0.54     18.21     0.10     2505     0.00     0.01  falk_dq_
  0.48     18.30     0.09     2261     0.00     0.00  cmprlb_
```

The dgemv routine is called a total of 10032 times and occupies 93% of the program runtime. Therefore, according to Amdahl, the theoretical maximum speed for this program would be 14× faster. In Chapter 7 it is shown that it is possible to come close to maximum theoretical speedup, and that this increase in speed scales well with problem size.

In this thesis, parallel programming novelties will be applied to an SAO program in an attempt to decrease the runtime. These are the four novelties:

1. Firstly, the inherent parallelism present by running 32-bit (single precision) routines on modern 64-bit architectures will be exploited;
2. Secondly, the problem dependency of dual and QP statements is investigated by running solvers in parallel;
3. Thirdly, the embarrassingly parallel nature of global optimization algorithms is advanced from a sequential to a parallel environment;
4. Lastly, the computational ability of modern graphics processing devices to solve highly parallel linear equations allows the replacement of computationally demanding linear algebra routines with routines optimized for highly parallel devices.

1.4 Chapter overview

Chapter 2 presents solution strategies for solving general nonlinear constrained optimization problems, with the use of the sequential approximate optimization (SAO) method.

Next, an in-depth overview of the function approximations is provided; in particular, convex separable (diagonal) quadratic approximations.

Lastly, the solution to the subproblems in dual and QP form is discussed.

Chapter 3 describes the different methods of parallelization used today and why OpenMP was chosen as the preferred language.

Chapter 4 provides a brief description of iterative refinement and mixed-precision algorithms, and it is discussed how these algorithms relate to the current study. These algorithms are then simplified to a simple update rule, which is then applied to a set of test problems, refining the single precision result to double precision accuracy. The results for each test problem are discussed.

Chapter 5 begins with a brief background to why dual and QP solvers may be used concurrently. Thread cancellation is then discussed and it is stated why it is applied using flags. The implementation is given and the results are documented for two test problems. Finally, some concluding remarks are given.

Chapter 6 shows that structural optimization function evaluations typically involve a complete finite element (FE) or boundary element (BE) analysis. By spreading the independent local minimization steps across multiple CPUs it becomes possible to determine the number of CPUs that will optimally reduce the cost of a multi-start global optimization implementation in a shared memory paradigm.

Chapter 7 studies the acceleration of approximation-based optimization methods by making use of accelerated dense matrix-vector products. These accelerated routines are based on the BLAS `dgemv` implementation, but are targeted to a specific hardware architecture, such as *multi-core* CPUs or *many-core* GPUs. Significant speedups are measured when evaluating the `dgemv` routine on the GPU – especially for large problem sizes.

Chapter 8 presents the conclusions and suggestions for future work.

Appendix A gives a brief history of precision and accuracy in numerical computation.

The difficulties in working with floating point values and how they affect scientific computing according to the IEEE 754 standard are further described.

Appendix B gives broad information on the computational devices used to solve the optimization problems. In addition, the optimization software and routines that are used to solve the set of design problems are discussed.

Appendix C provides an outline of the stopping criterion used in global optimization. The presentation in Snyman and Fatti (1987) is followed closely.

Appendix D demonstrates an OpenCL-mv implementation of the general matrix-vector product. This function is known in the BLAS standard library as dgemv (double precision general matrix-vector multiplication).

Appendix E This Appendix provides a listing of selected source code.

Appendix F Iteration histories are provided for the interested reader. These show how the problem properties change during execution and how the parallel algorithm described in Chapter 5 enables the SAO program to select the most effective solver in each step of the solution trajectory.

Chapter 2

Optimization

In this chapter, separable sequential approximate optimization (SAO) and sequential (diagonal) quadratic programming (SQP) methods are presented for solving general nonlinear inequality constrained optimization problems, as a basis for introducing the novelties mentioned in Section 1.2. In addition, function approximation, as well as an in-depth discussion of dual and quadratic approximate subproblems, is provided.

2.1 Sequential approximate optimization (SAO)

Consider the inequality constrained nonlinear optimization problem $\mathcal{P}$ of the form

$$\begin{aligned} \min_{\boldsymbol{x}} \ & f_0(\boldsymbol{x}) && \boldsymbol{x} = [x_1, x_2, ..., x_n]^T \in \mathcal{X} \subset \mathcal{R}^n \\ \text{subject to } & g_j(\boldsymbol{x}) \leq 0, && j = 1, 2, \ldots, m, \\ & \check{x}_i \leq x_i \leq \hat{x}_i, && i = 1, 2, \ldots, n. \end{aligned} \tag{2.1.1}$$

As mentioned in the introduction, $f_0(\boldsymbol{x})$ is the real valued scalar objective function, and the $g_j(\boldsymbol{x})$, $j = 1, 2, \ldots, m$ are m inequality constraint functions. $f_0(\boldsymbol{x})$ and $g_j(\boldsymbol{x})$ depend on the n real (design) variables $\boldsymbol{x} = [x_1, x_2, \ldots, x_n]^T \in \mathcal{X} \subset \mathcal{R}^n$. $\check{x}_i$ and $\hat{x}_i$ indicate lower and upper bounds on the primal variable x_i respectively. Additionally, functions $g_j(\boldsymbol{x})$, $j = 0, 1, 2, \ldots, m$ are assumed to be (at least) once continuously differentiable.

Both sequential approximate optimization (SAO) and sequential quadratic programming (SQP), as solution strategies for problem (2.1.1), seek to construct successive approximate analytical subproblems $\tilde{\mathcal{P}}[k]$, $k = 1, 2, 3, \ldots$ at successive approximations $\boldsymbol{x}^{\{k\}}$ to the solution $\boldsymbol{x}^*$.

The solution of subproblem $\tilde{\mathcal{P}}[k]$ is $\boldsymbol{x}^{\{k*\}} \in \mathcal{X} \subset \mathcal{R}^n$, which is obtained by using any suitable continuous programming method. Thereafter, $\boldsymbol{x}^{\{k+1\}} = \boldsymbol{x}^{\{k*\}}$, which is the minimizer of subproblem $\tilde{\mathcal{P}}[k]$. SAO and SQP methods importantly differ with respect to the approximations used. SQP methods construct quadratic approximations to the objective function f_0, and linear approximations $\bar{g}_j$ to the constraints. In addition, in SQP methods, the nonlinear curvatures of the constraints are incorporated by using the Hessian of the Lagrangian, rather than the Hessian of the objective function f_0 in the sequence of quadratic programs (QPs). SAO methods instead

are able to utilize a multitude of approximations, each of which is optimized for different sets of problems. However, all the approximations are separable.

Since our approximated problem $\tilde{\mathcal{P}}[k]$ can only be expected to be accurate over a small region of the allowable search domain, it is important that a trust region is placed around each new construction point to limit the step size. Here,

$$\check{x}_i \leftarrow \check{x}_i^{\{k+1\}} = \max(x_i^{\{k*\}} - \delta, \check{x}_i),$$
$$\hat{x}_i \leftarrow \hat{x}_i^{\{k+1\}} = \max(x_i^{\{k*\}} + \delta, \hat{x}_i),$$

where $x_i^{\{k*\}}$ is the solution to the subproblem at the kth iteration, $\check{x}_i$ is the prescribed lower bound and $\hat{x}_i$ is the prescribed upper bound on x_i. We will use default values for all test problems. δ is normally set so that $\delta = 0.2$, and $\check{x}_i$ and $\hat{x}_i$ are problem dependent and vary for all test problems. These values can be found in the initialization files for the respective problems.

Different approximations may be expected to be optimal for different problems. The simplest approach is to construct a linear approximation based on a Taylor series expansion. Other approximations that are better than the linear approximation are the reciprocal approximation, the exponential approximation and the approximations based on the popular method of moving asymptotes (MMA) – see Groenwold *et al.* (2007) and Svanberg (1987). To achieve better accuracy one could find intervening variables that make the approximated function behave more linearly, as is described in depth in Vanderplaats (2001) and Haftka and Gürdal (1992). Otherwise, the accuracy of the approximation can be increased by retaining higher order terms in the Taylor series expansion. Groenwold *et al.* (2010) give further details on approximation functions well suited for SAO routines. In the following sections, the primal, dual and QP subproblems are defined; these arise from diagonal quadratic approximations.

The flow diagram in Figure 2.1 shows the general operation of an SAO routine. Often, much computational expense goes into solving the subproblems as well as evaluating the function values. Hence, most of our focus will lie on these two aspects of SAO.

2.2 Function approximation

The success of SAO is highly dependent on the quality of the analytical approximations to the true objective and constraint functions. Retaining higher order terms of the Taylor series expansion increases the accuracy of the approximation, but it needs to be kept in mind that the calculation of the higher order terms increases the computational expense of the approximation; the storage requirements are of the order $O(n^2)$ for the quadratic information.

2.2.1 Diagonal quadratic approximations

The approximate subproblems herein are collected from Groenwold and Etman (2008), Etman *et al.* (2009) and Groenwold *et al.* (2010). Approximations $\tilde{f}(\boldsymbol{x})$ to the objective function $f_0(\boldsymbol{x})$

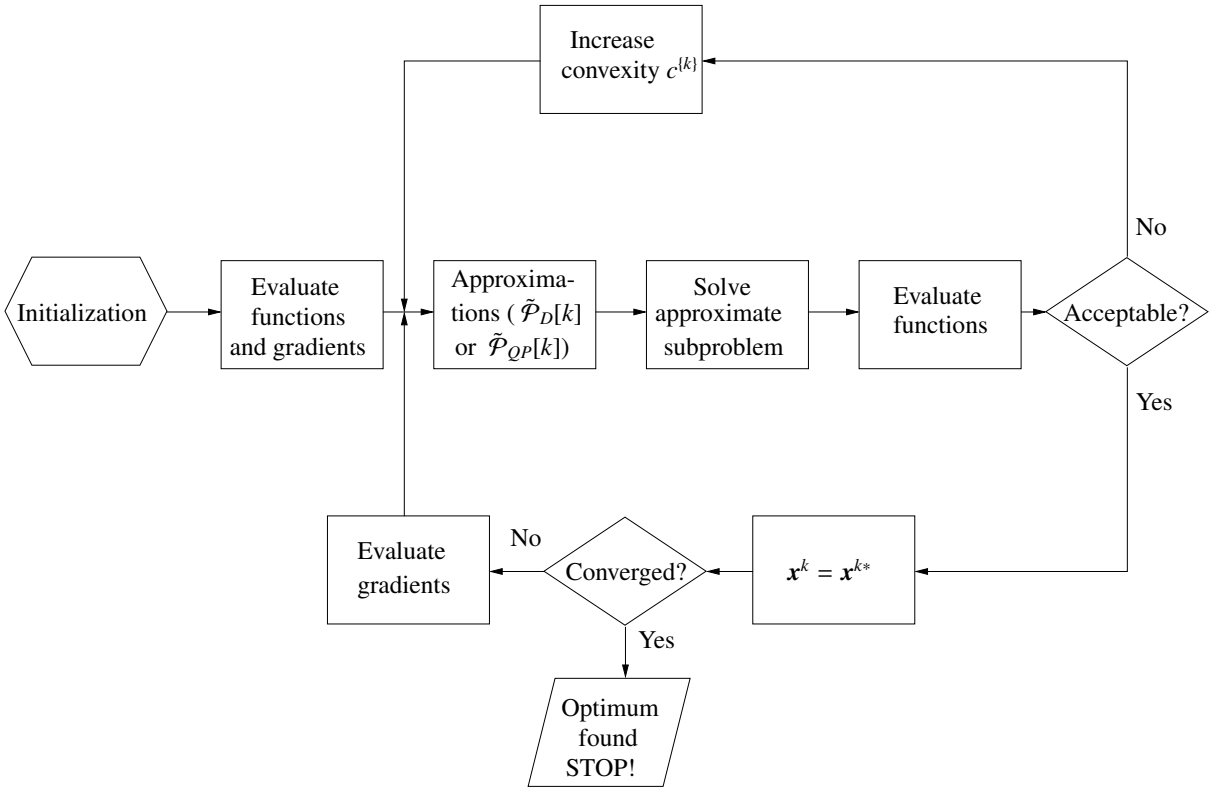


Figure 2.1: The simplified flow diagram of the SAO framework shows where approximations are made and subproblems are solved.

and all the constraint functions $g_j(\boldsymbol{x})$ are constructed such that

$$\tilde{g}_j(\boldsymbol{x}) = g_j^{\{k\}} + \boldsymbol{\nabla}^T g_j^{\{k\}} \boldsymbol{s} + \frac{1}{2} \boldsymbol{s}^T \boldsymbol{C}_j^{\{k\}} \boldsymbol{s}, \tag{2.2.1}$$

with $\boldsymbol{s} = (\boldsymbol{x} - \boldsymbol{x}^{\{k\}})$ and $\boldsymbol{C}^{\{k\}}$ an appropriate approximate *diagonal* Hessian matrix, for $j = 0, 1, 2, \ldots, m$. The abbreviated notation

$$g_j^{\{k\}} = g_j(\boldsymbol{x}^{\{k\}}), \tag{2.2.2}$$

etc. will be used as shorthand. For clarity, (2.2.1) is rewritten, using summation convention, as

$$\tilde{g}_j(\boldsymbol{x}) = g_j^{\{k\}} + \sum_{i=1}^{n} \left(\frac{\partial g_j}{\partial x_i} \right)^{\{k\}} (x_i - x_i^{\{k\}}) + \frac{1}{2} \sum_{i=1}^{n} c_{2i_j}^{\{k\}} (x_i - x_i^{\{k\}})^2, \tag{2.2.3}$$

emphasizing that the *approximate* second order Hessian terms or curvatures $c_{2i_j}^{\{k\}}$ are diagonal. To ensure strict convexity of each and every subproblem $\tilde{\mathcal{P}}[k]$, considered in sections to follow, convexity is enforced by

$$\begin{aligned} c_{2i_0}^{\{k\}} &= \max(\epsilon_0 > 0, c_{2i_0}^{\{k\}}), \\ c_{2i_j}^{\{k\}} &= \max(\epsilon_j \geq 0, c_{2i_j}^{\{k\}}), \qquad j = 1, 2, \ldots, m. \end{aligned} \tag{2.2.4}$$

ϵ_j, $j = 0, 1, 2, \ldots, m$ are prescribed at initialization, and are normally small. In other words, the approximate objective function $\tilde{f}_0$ is strictly convex, while the approximate constraint functions $\tilde{g}_j$, $j = 1, 2, \ldots, m$ are convex or strictly convex.

2.2.2 An approximate primal subproblem

From the diagonal quadratic approximations (2.2.3), the approximate continuous primal subproblem of $\tilde{\mathcal{P}}[k]$ at $\boldsymbol{x}^{\{k\}}$ is written as

Primal approximate subproblem $\tilde{\mathcal{P}}_p[k]$

$$\begin{aligned} \min_{\boldsymbol{x}} \ & \tilde{f}_0(\boldsymbol{x}) && \boldsymbol{x} \in \mathcal{X} \subset \mathcal{R}^n \\ \text{subject to } & \tilde{g}_j(\boldsymbol{x}) \leq 0, && j = 1, 2, \ldots, m, \\ & \check{x}_i \leq x_i \leq \hat{x}_i, && i = 1, 2, \ldots, n. \end{aligned} \tag{2.2.5}$$

This *primal approximate* problem contains n unknowns, m constraints, and $2n$ side or bound constraints (when no slack or relaxation variables are introduced); it may be solved using any technique for constrained nonlinear programming (dual and QP solvers are used). Other examples include penalty-based sequential unconstrained minimization techniques (SUMT), which seem rather cumbersome when high precision is desired, or augmented Lagrangian methods, etc.

2.2.3 An approximate dual subproblem

Instead of the primal approximate subproblem of $\tilde{\mathcal{P}}[k]$ for $k = 1, 2, 3, \ldots$, a *dual* approximate subproblem may be formulated. For each iteration k, the approximate *Lagrangian* $\mathcal{L}^{\{k\}}$ are defined as

$$\mathcal{L}^{\{k\}} = \tilde{f}_0(\boldsymbol{x}^{\{k\}}) + \sum_{j=1}^{m} \lambda_j^{\{k\}} \tilde{g}_j(\boldsymbol{x}^{\{k\}}), \tag{2.2.6}$$

where $\lambda_j^{\{k\}}$, also represented as a column vector $\boldsymbol{\lambda}$, represents the Lagrangian multipliers. Also note that the side constraints will be accommodated by the closed convex set over which the Falk dual is defined.

If the primal approximate subproblem is strictly convex, as it has been ensured in (2.2.4), then the stationary saddle point $(\boldsymbol{x}^*, \boldsymbol{\lambda}^*)$ defines the global minimizer of $\mathcal{L}^{\{k\}}$, with $\boldsymbol{x}^*$ the solution of the associated primal approximate subproblem. *I.e.*, $\mathcal{L}^{\{k\}}(\boldsymbol{x}^*, \boldsymbol{\lambda}^*) = \tilde{f}^{\{k\}}(\boldsymbol{x}^*)$, if and only if the KKT conditions are satisfied (Nocedal and Wright, 2006).

Therefore the dual function $\tilde{\gamma}(\boldsymbol{\lambda})$ is defined as

$$\tilde{\gamma}(\boldsymbol{x}, \boldsymbol{\lambda}) = \min_{\boldsymbol{x}} \{\tilde{f}_0(\boldsymbol{x}) + \sum_{j=1}^{m} \lambda_j \tilde{g}_j(\boldsymbol{x})\}. \tag{2.2.7}$$

An optimization problem is called separable if the objective function as well as all the constraints are separable. In other words, when each can be written as the sum of functions of the individual

variables $x_1, x_2, ..., x_n$, *e.g.* $f(\boldsymbol{x}) = f_1(x_1) + f_2(x_2) + ... + f_n(x_n) + C_f$ and $g(\boldsymbol{x}) = g_1(x_1) + g_2(x_2) + ... + g_n(x_n) + C_g$. Separability is enabled by retaining only (approximate) diagonal terms in the Hessian matrix. Then, the separable Lagrangian, equation (2.2.7), can be rewritten as a function of $\boldsymbol{x}(\boldsymbol{\lambda})$. Let us denote the minimizer of equation (2.2.7) for $\boldsymbol{\lambda}$ given by $\boldsymbol{x}(\boldsymbol{\lambda})$. It may therefore be written

$$\tilde{\gamma}(\boldsymbol{\lambda}) = \tilde{f}_0(\boldsymbol{x}(\boldsymbol{\lambda})) + \sum_{j=1}^{m} \lambda_j \tilde{g}_j(\boldsymbol{x}(\boldsymbol{\lambda})), \tag{2.2.8}$$

which allows for the formulation of the dual approximate subproblem $\tilde{f}_D[k]$ as

Dual approximate subproblem $\tilde{\mathcal{P}}_D[k]$

$$\begin{aligned} &\max_{\boldsymbol{\lambda}} \{\tilde{\gamma}(\boldsymbol{\lambda}) = \tilde{f}_0(\boldsymbol{x}(\boldsymbol{\lambda})) + \sum_{j=1}^{m} \lambda_j \tilde{g}_j(\boldsymbol{x}(\boldsymbol{\lambda}))\}, \\ &\text{subject to } \lambda_j \geq 0, \qquad j = 1, 2, \ldots, m. \end{aligned} \tag{2.2.9}$$

This bound constrained problem requires the determination of the m unknowns λ_j only, subject to m non-negativity constraints on the λ_j.

If primal subproblem (2.2.5) is strictly convex and separable, and the approximation functions f_0 and g_j are simple enough, it may be possible to find a simple (analytical) expression for the minimizing primal variables $\boldsymbol{x}(\boldsymbol{\lambda})$ in (2.2.8) in terms of the dual variables $\boldsymbol{\lambda}$. In many cases it does not make sense to solve (2.2.9) instead of the primal (2.2.5), since it may require more computational work to obtain a nested optimization problem. However, since the objective function and constraints are *separable*, the maximization of equation (2.2.9) and the minimization of (2.2.7) become simple to execute (Nocedal and Wright, 2006). For the diagonal quadratic approximations (2.2.3), this is indeed the case.

After deriving $\dfrac{\partial \gamma^2(\lambda)}{\partial x_i}$ in terms of x_i, introducing the curvatures in equation (2.2.4) and rearranging, we obtain

$$\beta_i(\boldsymbol{\lambda}) = x_i^{\{k\}} - \left(c_{2i_0}^{\{k\}} + \sum_{j=1}^{m} \lambda_j c_{2i_j}^{\{k\}} \right)^{-1} \left(\frac{\partial f_0^{\{k\}}}{\partial x_i} + \sum_{j=1}^{m} \lambda_j \frac{\partial f_j^{\{k\}}}{\partial x_i} \right), \tag{2.2.10}$$

with

$$x_i(\boldsymbol{\lambda}) = \begin{cases} \beta_i(\boldsymbol{\lambda}) & \text{if} \quad \check{x}_i < \beta_i(\boldsymbol{\lambda}) < \hat{x}_i, \\ \check{x}_i & \text{if} \quad \beta_i(\boldsymbol{\lambda}) \leq \check{x}_i, \\ \hat{x}_i & \text{if} \quad \beta_i(\boldsymbol{\lambda}) \geq \hat{x}_i, \end{cases} \tag{2.2.11}$$

for $i = 1, 2, \ldots, n$, being the final result that is required for solving the dual problem (2.2.9) with the diagonal quadratic approximations (2.2.3). The optimal point in the primal space of subproblem k is denoted $\boldsymbol{x}^{\{k*\}}$, which will conditionally converge to $\boldsymbol{x}^*$, the minimizer of the primal problem.

As said, the dual approximate subproblem (2.2.9) requires the determination of the m unknowns λ_j only, subject to m non-negativity constraints on the λ_j. This seems particularly advantageous when $m \ll n$. The current implementation uses a limited memory BFGS variable metric solver

(Zhu *et al.*, 1994and Byrd *et al.*, 1995), which is able to take the simple non-negativity constraints into consideration. To do so, it is only required to calculate the derivatives with respect to the dual variables $\boldsymbol{\lambda}$, which are obtained as

$$\frac{\partial \tilde{\gamma}}{\partial \lambda_j} = \tilde{g}_j(\boldsymbol{x}(\boldsymbol{\lambda})), \quad j = 1, 2, \ldots, m \tag{2.2.12}$$

which are the nonlinear constraint approximations. If the subproblems happen to be infeasible, a bounded form (Wood *et al.*, 2010) of the dual approximate subproblem $\tilde{\mathcal{P}}_D[k]$ is used, but relaxation (see Svanberg (2002), Groenwold *et al.* (2009) and Svanberg (1987)) may equally well be used. For further details on diagonal quadratic approximations and the dual by Falk, the reader is referred to Falk (1967) and Groenwold and Etman (2008).

2.2.4 Approximate subproblem in QP form

Since the approximations (2.2.1) are (diagonal) quadratic, the subproblems are easily recast as a quadratic program $\tilde{\mathcal{P}}_{QP}[k]$, written as

Dual approximate subproblem $\tilde{\mathcal{P}}_{QP}[k]$

$$\begin{aligned} \min_{\boldsymbol{s}} \ & \bar{f}_0^{\{k\}}(\boldsymbol{s}) = f_0^{\{k\}} + \boldsymbol{\nabla}^T f_0^{\{k\}} \boldsymbol{s} + \frac{1}{2} \boldsymbol{s}^T \boldsymbol{Q}^{\{k\}} \boldsymbol{s} \\ \text{subject to } & \bar{f}_j^{\{k\}}(\boldsymbol{s}) = g_j^{\{k\}} + \boldsymbol{\nabla}^T f_j^{\{k\}} \boldsymbol{s} \leq 0, \qquad j = 1, 2, \cdots, m, \\ & \check{x}_i \leq x_i \leq \hat{x}_i, \qquad i = 1, 2, \cdots, n, \end{aligned} \tag{2.2.13}$$

with $\boldsymbol{s} = (\boldsymbol{x} - \boldsymbol{x}^{\{k\}})$ and $\boldsymbol{Q}^{\{k\}}$ the Hessian matrix of the approximate Lagrangian.

Using the diagonal quadratic objective function and constraint function approximations $\tilde{g}_j$, $j = 0, 1, \ldots, m$, the approximate Lagrangian $\mathcal{L}^{\{k\}}$ is given in (2.2.6). This gives diagonal terms

$$Q_{ii} = c_{2i_0}^{\{k\}} + \sum_{j=1}^{m} \lambda_j^k c_{2i_j}^{\{k\}}, \tag{2.2.14}$$

and remaining off-diagonal terms $Q_{il} = 0 \ \forall \ i \neq l$, $i, l = 1, 2, \cdots, n$. As for the dual subproblem, convexity conditions (2.2.4) are applied to arrive at a strictly convex QP subproblem with a unique minimizer.

The quadratic programming problem requires the determination of the n unknowns x_i, subject to m linear inequality constraints. Because of the similarities in using this method in SAO and SQP, it is often referred to as SQP-like approximation. Efficient QP solvers can typically solve problems with very large numbers of design variables n and constraints m. Obviously, it is imperative that the diagonal structure of $\boldsymbol{Q}$ is exploited when the QP subproblems are solved. Herein, the commercial Numerical Algorithms Group (NAG) QP solver `E04NQF` (NAG Library Manual, Mark 22, 2009) and the `LSQP`solver from GALAHAD (Gould *et al.*, 2004) are used.

2.3 Which approximation method is best?

The simple answer to the above question would be: none. All approximation methods have their specific benefits for a certain class of problems. In their paper, *No Free Lunch Theorems for Optimization*, William G. Macready and David Wolpert describe the connection between effective optimization algorithms and the problems they are solving. A number of "no free lunch" (NFL) theorems are defined which establish that, for any algorithm, any superior performance for a certain class of problem is offset by the performance for another class. Therefore, it has become important to understand the relationship between how well an algorithm performs and the optimization problem $\mathcal{P}$ on which it is run (Macready and Wolpert, 1997). As argued, the problem-specific nature of structural optimization algorithms causes an approximation to be favoured by a specific problem type. Etman *et al.* (2009) have shown that dual approximations are particularly advantageous if $m \ll n$. However, SQP-like methods are still one of the most promising alternatives when used in combination with QP solvers when both n and m are large.

In Chapter 5 it will be shown that there is no need to rely on only one approximation and hence one type of solver, but that one can "order an entire buffet" of algorithms to do the work, hence decreasing solution time. With the addition of parallel algorithms, the actual computational effort, and with that also the useful work, should decrease.

Chapter 3

Parallel computing

This chapter summarizes some fundamentals of parallel programming and hardware, and gives a brief overview of some of the more common parallel programming models. Finally, the characteristics of each programming model is provided and it is stated why the respective software languages are used for the applications.

3.1 Types of parallelism

There are four major types of parallelism used in computer technology today, namely bit-level, instruction-level, data and task parallelism, as described below (Culler *et al.*, 1999):

Bit-level parallelism: Parallelism is achieved by doubling computer word size, *i.e.* the amount of information the processor can manipulate per cycle or word size. Increasing the word size reduces the number of instructions the processor must execute to perform an operation on variables whose sizes are greater than the length of the word. For example, suppose a 32-bit processor must add two 64-bit integers; first the processor must add the 32 lower-order bits from each integer using standard addition instructions, and then the 32 higher-order bits can be added using an add-with-carry instruction and the carry bit from the lower order addition. Therefore, two instructions are needed to complete a single operation on a 32-bit architecture, whereas a 64-bit processor would be able to complete the operation in a single instruction.

Instruction-level parallelism: This type of parallelism is the result of the re-ordering and combining of a computer program into groups of instructions, which are then executed in parallel without changing the result of the program. Instruction-level parallelism can only be achieved on processors that are pipelined. Modern processors have multi-stage instruction pipelines. A different task in an instruction can be performed for every stage in the pipeline. This means that a single processor with an N-stage pipeline can have up to N different running instructions, each at a different stage of completion.

Data parallelism: This type of parallelism is inherent in program loops, where data are distributed across different computing nodes to be processed in parallel. Parallelizing loops

often lead to similar (not necessarily identical) operation sequences or functions being performed on elements of large data structures.

Task parallelism: Task parallelism is the characteristic of a parallel program according to which entirely different calculations can be performed on either the same or different sets of data. This contrasts with data parallelism, where the same calculation is performed on the same or different sets of data. Task parallelism does not usually scale with the size of a problem.

These are the types of parallelism that make up computing systems. The following section provides a closer inspection of the hardware that is used to run parallel codes.

3.2 Hardware

The capability to perform multiple operations in parallel is highly dependent on the hardware architecture of the system. Let us now take a look at some of the traditional parallel hardware architectures and how they work.

3.2.1 Memory parallelism

Parallelism in the memory systems has the greatest effect on programming models and algorithms. Presented here is a brief overview of parallel memory systems. Culler *et al.* (1999) give a much more detailed discussion of parallel memory systems. There is one major choice when connecting two uniprocessors for parallel computation, that is, the choice between a distributed memory system and a shared memory system. However, these two memory paradigms can be combined into *distributed shared-memory* (DSM) systems.

Distributed memory: This is the simplest approach from a hardware perspective, because separate computers are connected via a network. The typical programming model consists of a separate process on each computer, and each computer communicates with the other by sending a message (*message passing*). Distributed memory systems have Non-uniform Memory Access (NUMA).

Shared memory: This approach ties the computers more closely. All the memory is placed into a single address space. That is, data are available to all CPUs for load and store instructions. Each element of main memory can be accessed with equal latency and bandwidth. These are known as Uniform Memory Access (UMA) systems. This method has higher bandwidth and lower latency for memory access than distributed memory systems. Two major issues affect these systems, namely memory consistency and coherence. These make it more difficult for the programmer to write efficient code (Dongarra *et al.*, 2003).

Distributed shared-memory: These hybrid systems allow a processor to directly access a datum in remote memory. On these *distributed shared-memory* (DSM) systems, the latency associated with a load varies with the distance to remote memory. The direct access to

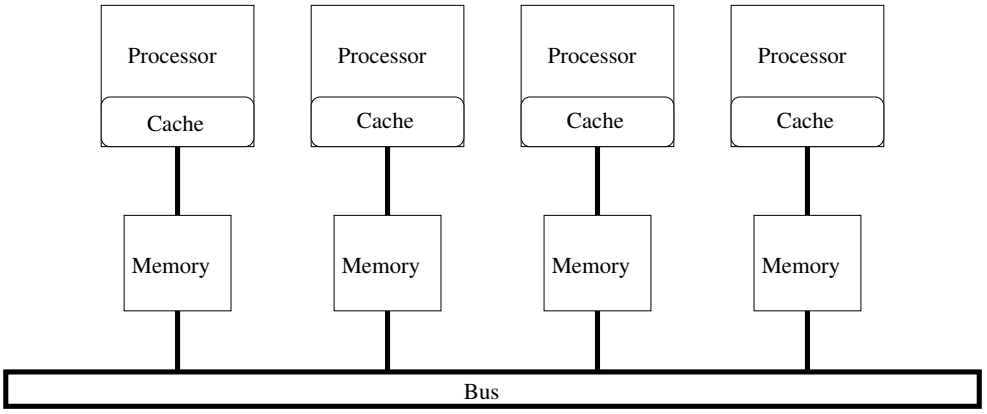


Figure 3.1: Non-uniform access shared-memory architecture or distributed memory.

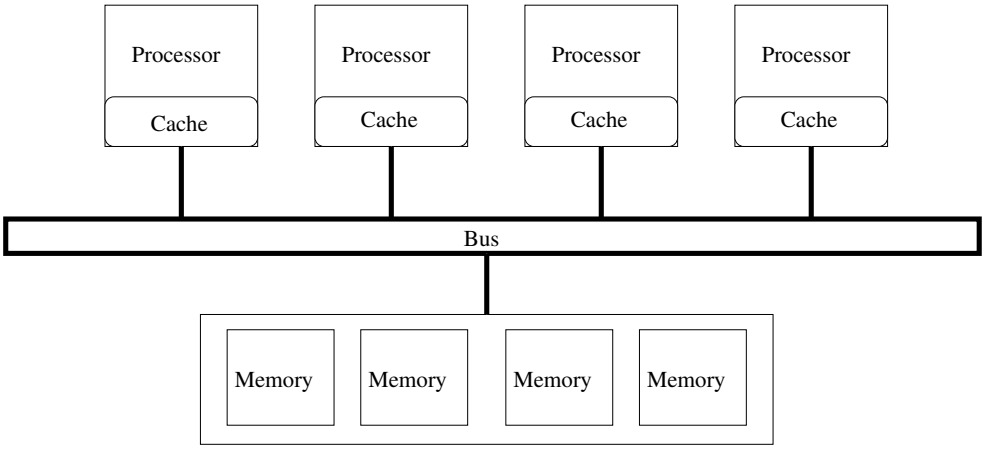


Figure 3.2: Uniform access shared-memory architecture.

remote memory is handled by sophisticated network interface units that ensure cache coherency. Computer systems make use of caches – small, fast memories located close to the processor that store temporary copies of memory values. A cache coherency system that keeps track of cached values and strategically purges them, thus ensuring correct program execution (Dongarra *et al.*, 2003).

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism. This classification is broadly analogous to the distance between basic computing nodes. Listed here are the classes of parallel architectures available today.

- *Multicore computing* – A multicore processor is a processor that includes multiple execution units, or "cores", on the same chip.
- *Symmetric multiprocessing* – A symmetric multiprocessor (SMP) is a computer system with multiple identical processors that share memory and connect via a bus.

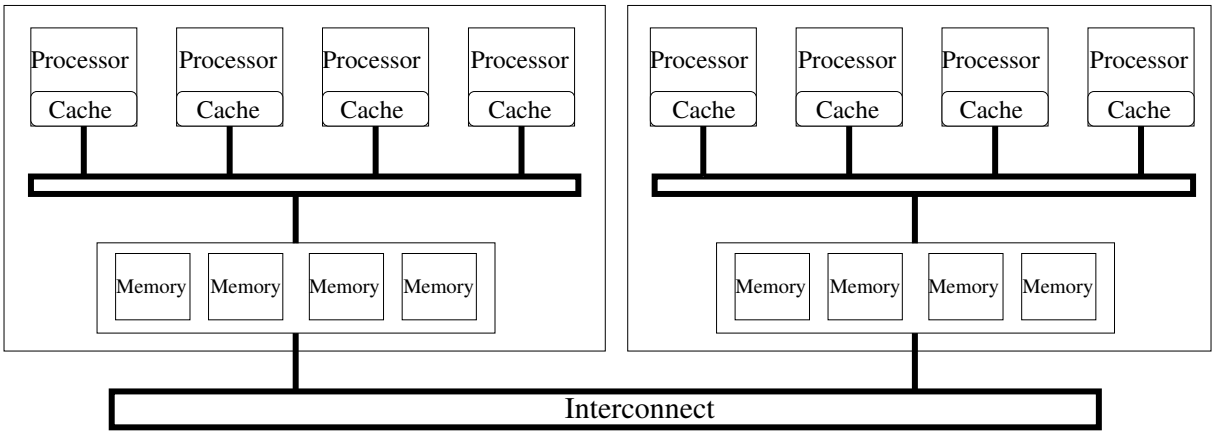


Figure 3.3: Hybrid distributed shared-memory architecture.

- *Distributed computing* – A distributed computer (also known as a distributed memory multiprocessor) is a distributed memory computer system in which the processing elements are connected by a network. These include cluster computing, massive parallel processing and grid computing.
- *Specialized parallel computers* – Within parallel computing, there are specialized parallel devices that remain niche areas of interest. While not domain-specific, they tend to be applicable to only a few classes of parallel problems. These are: field-programmable gate arrays, graphics processing units (GPUs), integrated circuits and vector processors. The most successful of these are GPUs, which are slowly starting to dominate the field of data parallel operations, in particular linear algebra matrix operations.

One must not only consider the hardware, but also choose the appropriate programming model that will best suit your hardware. This choice will affect the choice of programming language system and library for implementation of the application. This choice is further discussed in the next section on programming models.

3.3 Programming models

When creating parallel programs, two dominant alternatives arise: *message passing* and *multithreading*. These two approaches can be distinguished in terms of how concurrently executing segments of an application share data and synchronize execution.

- **Message passing:** data are shared by the explicit copying, or "sending", from one parallel thread or process to another. Thread synchronization is implicitly handled with completion of the copy directive.
- **Multithreading:** data are implicitly shared through the use of shared memory. Synchronization must be handled explicitly by mechanisms such as locks, semaphores, and condition variables.

The advantages and disadvantages are obvious, as multithreading allows programs to execute particularly efficiently on computers utilizing physically shared memory. However, for the reason of scalability, many parallel computers built today do not support shared memory across the the entire system. If this is the case, hybrid multithreaded-message passing approaches (Dongarra *et al.*, 2003) is more appropriate.

Another fairly recent trend in computer engineering research is general-purpose computing on graphics processing units or GPGPU-computing. GPUs are co-processors that have been heavily optimized for computer graphics processing. However, these are a subset of multithreading, and this technology is discussed further in Section 7.2 because of definitive architectural differences between CPUs and GPUs. Although GPGPU technologies have dominated computer graphics processing for many years, they are increasingly being used for non-graphic-based linear algebra matrix operations. Today, GPUs are especially well suited for highly parallel applications that are computationally intensive and have highly regular memory-access patterns (Boggan and Pressel, 2007).

3.4 Application programming interfaces (APIs)

In order so take advantage of the computational abilities of the above-mentioned parallel architectures, the programmer needs the appropriate application programming interface (API). Some of the software technologies that arose from these approaches to parallel programming are summarized below.

3.4.1 Message passing interface (MPI)

Message passing is by far the most widely used approach to parallel computing, dominating most high performance computing (HPC) systems because of its scalability. The official standard, called the Message Passing Interface (MPI Forum, 1994 and Snir *et al.*, 1998), was first compiled by a group called the MPI Forum. The standard itself is available on the web at `http://www.mpi-forum.org`.

MPI is a rich and sophisticated library including a wide variety of features. Although MPI is a complex, multifaceted system, only a general overview is given; for more complete information, discussions and tutorials, the reader is referred to Gropp *et al.* (1999*b*), Gropp *et al.* (1999*a*), Foster and Kesselman (1997) and Pacheco (1997).

According to the message-passing model, processes communicate by calling library routines to send and receive *messages*. This type of communication is known as *cooperative* communication. Data are sent by a calling routine and are only received once the destination routine calls a receiving routine. Dongarra *et al.* (2003) describe six MPI functions that are used the most widely to solve a wide range of problems. These functions are used to initiate and terminate a computation, identify processes, and send and receive messages:

MPI_INIT:	Initiate an MPI computation.
MPI_FINALIZE:	Terminate a computation.

MPI_COMM_SIZE:	Determine number of processes.
MPI_COMM_RANK:	Determine my process identifier.
MPI_SEND:	Send a message.
MPI_RECV:	Receive a message.

General MPI program structure starts with the MPI-include file, which for FORTRAN would be `include 'mpif.h'`. Next, one would have some declarations and/or prototypes before program executions starts. Normally there is some serial code before the parallel code begins. This is done with a call to `MPI_INIT`. Depending on the parallel operations required within the body of the parallel code, *point-to-point* communication is initialized by *send* and *receive* operations running on concurrently executing program components. *Collective* operations, such as broadcast and reductions, are often used to implement common global operations involving multiple processes (Dongarra *et al.*, 2003). The parallel environment ends with a call to `MPI_FINALIZE`, and serial code continues until program execution terminates.

MPI models also differ in a variety of ways, affecting the way threads are synchronized and how data are handled on arrival at the thread. For example, the send and receive calls may be blocking or non-blocking; the means with which send and receive calls are matched up may also differ. These types of variations in message passing have a significant impact on the performance. Three major factors influence performance (Dongarra *et al.*, 2003):

1. Bandwidth: The bandwidth achieved by a specific implementation is often dominated by the amount of time the data must be copied during data transfer operations between application components. Poorly designed interfaces could result in excess copies, reducing the overall performance.

2. Latency: Latency is dominated by the message setup time rather than the actual message "flight" time across the network. Thus overhears incurred by initializing buffers and interfacing with hardware may be significant.

3. Message passing: Message passing deals with overlapping communication and computation. For example, non-blocking sending enables the sender to continue execution even if all the data have not yet been accepted by the receiver, whereas non-blocking receives enable the receiver to anticipate the next incoming data elements, while still performing work. In both situations the resulting application is improved.

The message-passing model does, however, reveal two great strengths. The most obvious is its high portability. The second strength is the explicit control over the memory used by each process. Since memory access and placement often determine performance, the ability to manage memory access and position can allow the programmer to achieve high performance. However, this high performance comes at a price. Highly efficient code in MPI requires highly skilled and experienced programmers, making MPI one of the most difficult implementations to program.

3.4.2 OpenMP

OpenMP is based on the multithreaded programming model and provides a portable, scalable model for developers of shared memory parallel applications. After a first attempt at a standard in 1994, ANSI X3H5 was drafted. As this standard never was adopted due to the newer shared memory machine architectures becoming more prevalent, OpenMP took over the reigns from ANSI X3H5 and the OpenMP standard specification came to life in 1997 after the formation of the Architecture Review Board (ARB) (Chapman *et al.*, 2008).

OpenMP is a shared-memory application programming interface (API). This model assumes that all concurrently executing program components share a single common address space. No special operations are needed for copying information, and program components can exchange information simply by reading and writing to memory with normal operations. Because the address space is shared between concurrent processes, we refer to these processes as *threads*; hence the name *multithreaded programming* (Dongarra *et al.*, 2003).

As depicted by the so-called fork-join programming model (Dennis and Horn, 1966), the program starts as a single thread of execution. The thread that executes this code is referred to as the *initial thread*. After an OpenMP parallel construct is encountered, a team of threads is created (the *fork*). The initial thread becomes the master and, at the end of the construct, only the original thread, or master of the team, continues; all others terminate (the *join*).

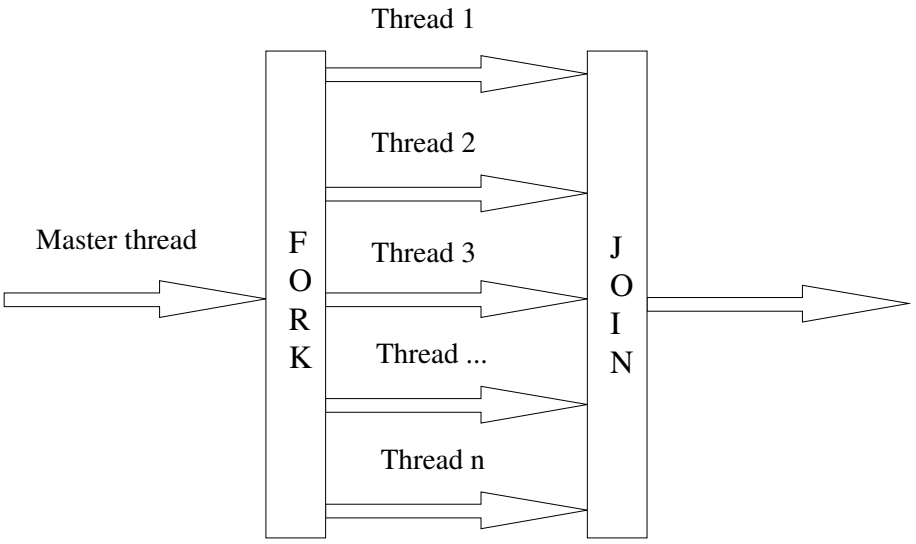


Figure 3.4: The fork-join model of parallel execution in OpenMP.

Work is shared amongst the threads with the use of OpenMP work-sharing *directives*. These directives instruct the compiler to create threads, perform synchronization operations and manage shared memory. It is up to the programmer to specify how work is to be shared among the executing threads. If not explicitly overridden by the programmer, there exists an implicit synchronization barrier at the end of all work-sharing constructs. The choice of work-sharing method may have a considerable effect on the performance of the program.

Following the fork-join model, the use of threads in OpenMP is highly structured, because OpenMP was designed specifically for high performance parallel applications. By being specifically focused on the needs of parallel programs, OpenMP can result in a more convenient and higher-performance implementation of multithreaded programs.

3.4.3 POSIX threads

POSIX is an acronym for Portable Operating System Interface. Although the acronym originated to refer to the original IEEE Standard 1003.1-1988, the name POSIX is now used to refer to a family of related standards: IEEE Std 1003.n (where n is a number) and the parts of ISO/IEC 9945.

This standard was required after code portability became a concern for software developers. Because hardware vendors were implementing their own proprietary versions of threads, software developers were unable to take full advantage of the capabilities provided by threads. Thus, a standardized C language threads programming interface was specified, for UNIX systems, namely the IEEE POSIX 1003.1c standard. Implementations that adhere to this standard are often also referred to as Pthreads. The POSIX standard has continued to evolve and undergo revisions. The latest version is known as IEEE Std 1003.1 (POSIX, 2004).

Table 3.1, from Barney (2008), compares timing results for the `fork()` subroutine and the `pthread_create()` subroutine. Timings reflect 50000 process or thread creations. No optimization flags were used and timing values were logged using the UNIX time utility.

Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
AMD 2.3 GHz Opteron	1.8	5.3
AMD 2.4 GHz Opteron	1.2	5.3
IBM 1.9 GHz POWER5 p5-575	4.1	16
IBM 1.5 GHz POWER4	2.1	4
Intel 2.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

Table 3.1: Comparison of several platforms' memory-to-CPU bandwidth copy results (Barney, 2008).

What makes Pthreads so advantageous is that threads within a process share the same address space, just like OpenMP, therefore inter-thread communication is more efficient and, in many cases, easier to use than inter-process communication. Other than using on-node task communication via shared memory, such as MPI, the threads of a Pthreads process share the same address space, eliminating intermediate memory copy operations. Therefore, most memory operations are cache-to-CPU, or memory-to-CPU in the worst case scenario. This potential performance increase is shown in Table 3.1. However, Pthreads has very important uses outside of parallel programming for high performance applications. Multithreading proves to be very effective

when asynchronous requests need to be handled (Dongarra *et al.*, 2003). For this reason, the POSIX operating system interface includes a thread library. Although many of the features of POSIX threads are not of interest to parallel program developers, this ability to handle asynchronous requests makes it ideal for the intended purposes of this study. However, Pthreads is highly platform dependent and is mostly aimed at codes written in C and C++ (see Barney, 2008).

3.4.4 CUDA

CUDA (Compute Unified Device Architecture), the computing architecture developed by Nvidia, gives programmers access to the virtual instruction set and memory of the parallel computational elements on CUDA-enabled GPUs (graphics processing units). Unlike CPUs, GPUs have a parallel throughput architecture that emphasizes the execution of many concurrent threads slowly, rather than executing a single thread very quickly. This makes GPUs ideally suited for solving SIMD algorithms e.g. large linear-algebraic matrix and vector calculations where the high computation densities can most often be evaluated independently. This approach to solving general purpose problems on GPUs is known as GPGPU computing.

Due to the increasing popularity of using GPUs for the acceleration of non-graphical applications, as used in scientific calculations, the CUDA architecture now shares a range of computational interfaces with its two competitors - the Khronos Group's Open Computing Language (OpenCL) (Khronos Group [Online], 2009) and the most recent, Microsoft's DirectCompute (NVIDIA Corporation, 2011).

3.4.5 OpenCL

OpenCL (Open Computing Language) is a framework for writing programs that execute in concert across heterogeneous platforms consisting of x86 core CPUs and GPUs. OpenCL is managed by the non-profit technology consortium, the Khronos Group.

OpenCL is analogous to the open industry standards, OpenGL and OpenAL, for 3D graphics and computer audio respectively. OpenCL uses kernels (functions that execute on OpenCL devices) to control execution on the respective platforms, whereas the host program that executes on the host system sends kernels to execute on OpenCL devices, using a command queue (Advanced Micro Devices, Inc., 2010*b*).

Task-based and data-based parallelism may be applied in a wide variety of computing applications. It has therefore been adopted into graphics card drivers by both AMD/ATI and Nvidia.

3.5 Which API fits the design problem best?

MPI suffers from the same issues as OpenMP, as completion of a thread (a call to `MPI_FINALIZE`) should occur on the same thread that initialized the MPI environment. This *main thread* is only able to initiate the `MPI_FINALIZE` call once all the process threads have completed their MPI calls and have no pending communications or I/O operations (Message Passing Interface Forum,

1997). Hence, a thread cannot be cancelled from the outside and termination can only occur at predefined exit points, such as at the flag points in our OpenMP algorithm 2. This, combined with the difficulty in process control, scheduling and inter-thread communication makes MPI a bad choice for use in irregular algorithms.

Though there are some implementations of Pthreads for FORTRAN, in particular the one on the IBM(AIX) platform (non-portable), they are not widely available. Therefore, the only way these could be used is by writing wrappers over them. These wrappers need to be simple and succinct enough to be used in real applications. This has not been investigated in this thesis due to time constraints.

To summarize, OpenMP is used because of the simple workaround to thread cancellation with the use of flags, which is discussed further in Section 5.2.2. Although Nikolopoulos *et al.* (2001) developed the OMP*i* compiler with thread cancellation support for OpenMP, it is preferred not to use it because the OMPi compiler would need to be modified for use with Fortran codes. Similarly, wrapper functions would need to be used for the C codes in Pthreads. Lastly, MPI has obvious limitations in process control, scheduling and inter-thread communication. For example, in an iterative program, such as in SAO, if two irregular algorithms are to be started in parallel, the next iteration should start, once the first algorithm completes. However, the current thread must first communicate with the competing thread that it has completed. The competing thread must then first safely exit execution before the next iteration may start. Therefore, much scheduling and inter-thread communication is necessary to perform this one simple task.

There is much speculation about whether OpenCL or CUDA is better. CUDA is a more developed technology, whereas OpenCL is still very much in its infancy. OpenCL caters for a larger spectrum of devices, enabling kernels to execute on both multi-core CPUs and many-core GPUs. With AMD and Nvidia platforms delivering competing performance, it is hard to choose, although recent performance graphs show AMD consistently outperforming Nvidia. Chapter 7 uses CUDA's supplied `dgemv` routine to run benchmarks initial tests on Nvidia's CUDA architecture. Further, in Appendix D, a modified kernel developed by Bainville (2010) is used to run a simple `dgemv` experimental version on an AMD system.

Chapter 4

Single and double precision paradigms

This chapter takes a brief look at the solution methods for linear systems and how they relate to the current study. These methods are: direct methods, iterative refinement and mixed-precision algorithms. The methods are then reduced to a simple update rule and applied to a set of test problems. The iterative refinement method is mimicked by refining the single precision result to double precision accuracy by some update rule. All test problems are discussed in this chapter.

4.1 Background

Scientific computing has largely focused on double precision arithmetic in the past. Double precision (64-bit) execution units were fully pipelined[1] and capable of completing at least one operation per clock cycle. This meant that high-degree instruction-level parallelism was only possible by introducing more functional units and relatively expensive speculation mechanisms. Therefore, full hardware resource utilization was not guaranteed. The adoption of Single Instruction Multiple Data (SIMD) processor extensions in the mid-90s made it possible to benefit from the use of single precision arithmetic in some cases, where the higher accuracy of double precision was not required, making this a type of bit-level parallelism. Since the goal is to process an entire vector in one single operation, the computational throughput would double, while the data storage space would halve, when completing a 32-bit operation on an 64-bit processor. Today, there are two prominent examples of SIMD extensions, namely: streaming SIMD extensions (SSE2) for the AMD and the Intel line of processors and AltiVect for the PowerPC's Velocity Engine (Buttari and Dongarra, 2007).

Benchmarking analysis and architectural descriptions done by Buttari and Dongarra (2007); Buttari *et al.* (2008) reveal that, on many commodity processors, the performance of single precision computations (32-bit floating point arithmetic) may be significantly higher than that of double precision computations (64-bit floating point arithmetic). In the SSE2 case, a vector unit can complete four single precision operations every clock cycle, but only two in double precision. Similarly, for the AltiVec, single precision can complete eight floating point operations in

[1] Modern processors have multi-stage instruction pipelines. Each stage in the pipeline corresponds to a different action performed by the processor on that instruction in that stage; a processor with an N-stage pipeline can have up to N different instructions at different stages of completion.

single precision, but only four floating point operations in double precision. The results for these and other architectures are presented in Table 4.1. They all exhibit an overall single precision performance increase of approximately 2× over double precision computations.

	Size	SGEMM/DGEMM	Size	SGEMV/DGEMV
AMD Opteron 246	3000	2.00	5000	1.70
Sun UltraSparc-IIe	3000	1.64	5000	1.66
Intel PIII Copp.	3000	2.03	5000	2.09
PowerPC 970	3000	2.04	5000	1.44
IntelWoodcrest	3000	1.81	5000	2.18
Intel XEON	3000	2.04	5000	1.82
Intel Centrino Duo	3000	2.71	5000	2.21

Table 4.1: Performance comparison between single and double precision arithmetic for matrix-matrix and matrix-vector product operations (Buttari *et al.*, 2008).

Iterative schemes feed the results of one iterative step, as input, into the next step and possibly execute hundreds or even thousands of iterations before completion. Thus, there is no easy way to obtain reliable error bounds. These difficulties often require high-precision arithmetic in hardware processors aiming to attain higher accuracy of the final result. While this is true in many cases, in general there is no monotonic relation between the precision of the quantization and the final accuracy, *i.e.* increasing the computational precision can lower the accuracy of the result. In practice, increasing the precision is often successful, so that implementations tend to favour the highest precision format supported by the hardware, without much consideration of whether it is really necessary. Accordingly, high-precision formats are used extensively even in the non-critical stages of code. The introduction of low-precision computations here could lead to significant speedup (Strzodka and Göddeke, 2006*a*). In a later publication, Göddeke *et al.* (2007) illustrated that attempting to solve a linear system in single precision fails, while double precision succeeds in reducing errors according to FE theory. Therefore this behaviour is of particular importance, because the increased computational effort is wasted if increased computational precision is unnecessarily used.

Buttari *et al.* (2008) improved the performance of some linear algebra operations by exploiting single precision operations to perform most of the computationally expensive tasks, while resorting to double precision at critical stages in an attempt to provide full double precision accuracy. The iterative refinement described in Demmel (1997) and Higham (2002) has been applied successfully to the solution of dense linear systems by Langou *et al.* (2006). Their work in architectural and benchmarking theory on single and double precision computations are based on earlier work by Langou *et al.* (2006).

With the performance benefits that single precision computations promises to have over double precision; single precision algorithms are applied to large-scale structural optimization programs. The double precision QP solvers are replaced with single precision QP solvers. Single precision solvers should be less expensive than double precision solvers, with the limitation of

the accuracy of the single precision result. Updating the single precision result with further double precision iterations should then deliver the required double precision accuracy.

By performing the bulk of the iterations in single precision and updating the solution to double precision after some error bound has been reached, the efficiency of using single precision QP solvers as a method for solving large-scale structural optimization problems is investigated. In this chapter a collection of results for several structural optimization problems. The problems are solved using single and double precision subsolvers. Results are provided for the `LSQP` solver, a production QP solver developed by GALAHAD (Gould *et al.*, 2004) that is freely available for academic use, in both single and double precision. These results are compared with those of the commercial Numerical Algorithms Group (NAG) (NAG Library Manual, Mark 22, 2009) QP solver, `E04NQF`. The single precision subsolver is an evaluation version that has been adapted from the double precision version by the NAG team specifically for our use. Results are given for both the pure single precision case as well as the case where the single precision result is updated to double precision.

More background to this chapter can be found in Appendix A, in which information is provided on precision and accuracy in scientific computation. The tools that were used are described in the software section, Appendix B.

4.2 Solution methods for linear systems

Previous work in the field of computational precision can be divided into two categories. One set of work aims to maximize raw floating point performance (FLOPS) in an IEEE standard representation, whereas the other is focused on mixing different precision operations to find a solution with a higher specified accuracy.

In the first category, acceleration is attained mainly by exploiting wide-parallelism, deep-pipelining and efficient data-paths. Baboulin *et al.* (2008) gives examples of such work, which includes implementations of both direct methods and iterative methods for the solution of systems of linear equations.

In the second category, acceleration is attained mainly by exploiting mixed-precision operations. The key idea behind mixed-precision schemes is to perform as many operations as possible in single precision, and only at critical stages execute a number of crucial operations in the more expensive double precision (Strzodka and Göddeke, 2006*b*; Buttari and Dongarra, 2007; Buttari *et al.*, 2008).

4.2.1 Direct methods

The direct method for solving linear systems, be they dense or sparse, is to apply Gaussian elimination to the coefficient matrix A with the use of LU decomposition. Firstly, the coefficient matrix A is factored into the product of a lower triangular matrix L and an upper triangular matrix U. In order to improve numerical stability of the resulting factorization, $PA = LU$, where P is a permutation matrix, only partial row pivoting is generally applied. The solution for the system of linear equations can then be achieved by solving $Ly = Pb$ by backward substitution and then

$Ux = y$ by forward substitution. Due to round-off errors, the computed solution x carries a numerical error magnified by the condition number of the coefficient matrix A (Demmel, 1997).

4.2.2 Iterative refinement

The iterative refinement algorithm is an iterative process applied to the computed solution at each iteration in order to improve the computed solution. Demmel (1997) points out that the non-linearity of the round-off errors makes the iterative refinement process equivalent to Newton's method when applied to the function $f(x) = b - Ax$. Provided that the system is not too ill-conditioned, the algorithm produces a solution correct to the working precision.

4.2.3 Mixed precision

Newton's algorithm (Ypma, 1995) uses an iterative algorithm to compute the zero of a function $f(x)$. This iterative formula is given in Equation (4.2.1). With the use of such an iterative formula, double precision accuracy can be achieved by refining a single precision solution. These are termed mixed precision algorithms.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{4.2.1}$$

Generally, the starting point and the gradient at the point $f'(x)$ are calculated in single precision. The refinement process is then computed in double precision. As long as the refinement process is less expensive than the initial computation, double precision accuracy can be achieved at nearly the same speed as the computed single precision result.

4.3 Mixed precision approach to solving SAO(QP) subproblems

The updating of the single precision result method is referred to throughout as the mixed precision method, even though mixed precision algorithms are not used in the classical sense as described above. Buttari *et al.* (2008) give a more in-depth discussion. Our investigation into the possibility of decreasing the computation time necessary for solving QP subproblems to double precision accuracy starts here.

First, `E04NQF` is incorporated into the SAO environment. The double precision library routine is a production version that is available from the electronic NAG libraries. The single precision version of the `E04NQF` library routine is an evaluation version specially developed by the NAG technicians. Second, the `LSQP` routine is incorporated.

The approximations are made in double precision, and then the quadratic programming problems are recast to single precision for solution by the single precision solvers. Some information is lost during the conversion to single precision, but the more accurate approximations decrease the

chance for numerical inconsistencies. The relative error difference is tested for at each iteration. Then, the update rule enables the change from single to double precision.

The time and precision with which both the single and double precision variants solve the optimization problems are investigated. The convex and non-convex problems given in Section 4.4.1 are evaluated, each for increasing amounts of design variables. The execution times for all different design variables for the single and double precision solvers are logged. Finally, the accuracy, Karush Kuhn Tucker (KKT) conditions and amount of iterations from the respective runs enable a measure of how accurately the respective design problems are solved.

4.3.1 Stopping condition and update rule

In order to determine the stopping condition, the relative error of the norms between the computed single and double precision, results must be calculated. The relative error between the norms is a measure of the problems condition and stability. The error on the norms may vary significantly, depending on the various problems and problem sizes. Therefore it is necessary to calculate the tolerance on the norm according to

$$\frac{dbl(x_{norm}) - sgl(x_{norm})}{dbl(x_{norm})} = \delta_{norm}, \tag{4.3.1}$$

where the notation $dbl(x_{norm})$ represents the double precision floating-point variable of x_{norm}, and $sgl(x_{norm})$ represents the single precision variable of x_{norm}. δ_{norm} is the relative error. The relative error is then calculated for each problem size, and the stopping condition for the current problem is set to $max[\delta_{norm}]$.

As further elaborated on in Appendix A, accuracy is lost in every operation because of distortions of the data introduced by the data quantization and rounding. This results in numerical difficulties during the solution of optimization problems, where the problems are often numerically ill-conditioned or unstable. The relative error values, ϵ_x, of the norm must be chosen large enough to keep within the working accuracy of the problem. Therefore, the update rule looks as follows:

```
if (delxnormc.lt.1.d-2.and.delxnormc.gt.1.d-20) then
  sd = 2
endif
```

While `sd = 1` the single precision routines are evaluated, and setting `sd = 2` starts execution of the double precision routines. Once the relative error of the result becomes less than 1×10^{-2}, the remaining iterates are solved by the double precision routine. In this way the greatest number of iterations are solved using the single precision routine. The complete source code is included in Appendix E.1.

4.4 Results

First, the results for the pure single precision subsolvers are compared with the results for pure double precision subsolvers. This enables us to verify that valid results are obtained within the required accuracy. Similar to the mixed precision algorithms introduced in Section 4.2.3, an updating algorithm is used that updates the single precision results by performing additional double precision iterations. By minimizing the number of double precision iterations that need to be performed, the single precision result may be updated to double precision accuracy.

In Tables 4.2 to 4.10 below, n represents the number of design variables, m represents the constraints, k^* represents the total number of outer iterations, f_0^* represents the function values for both single and double precision, h^* represents the maximum constraint violation, r^* represents the Karush Kuhn Tucker (KKT) condition, and CPU is the total CPU run time.

Results are collected for the following design problems: Svanberg's non-convex test problems, the first and second problem, the cam design problem of Dolan *et al.* (2004), Vanderplaats' cantilever beam and a n-dimensional generalization of Svanberg's n-variate cantilever beam as by Toropov (Groenwold, 2008).

4.4.1 Svanberg's non-convex test problems

The two non-convex problems proposed by Svanberg (Svanberg, 2002) often appear in structural optimization, in particular in topology optimization. Both are expressed in terms of the symmetric, *fully populated* $n \times n$ matrices $\boldsymbol{S}$, $\boldsymbol{P}$ and $\boldsymbol{Q}$, with elements given by

$$s_{ij} = \frac{2 + \sin(4\pi\vartheta_{ij})}{(1 + |i - j|)\ln(n)}, \quad p_{ij} = \frac{1 + 2\vartheta_{ij}}{(1 + |i - j|)\ln(n)}, \quad q_{ij} = \frac{3 - 2\vartheta_{ij}}{(1 + |i - j|)\ln(n)},$$

where

$$\vartheta_{ij} = \frac{i + j - 2}{2n - 2} \in [0, 1] \; \forall \; i, j,$$

and $n > 1$.

4.4.1.1 Svanberg's first test problem

The first problem is formulated as follows

$$\begin{aligned} \min_{\boldsymbol{x}} \; & f_0(\boldsymbol{x}) = \boldsymbol{x}^T \boldsymbol{S} \boldsymbol{x}, \\ \text{subject to } & f_1(\boldsymbol{x}) = \frac{n}{2} - \boldsymbol{x}^T \boldsymbol{P} \boldsymbol{x} \leq 0, \\ & f_2(\boldsymbol{x}) = \frac{n}{2} - \boldsymbol{x}^T \boldsymbol{Q} \boldsymbol{x} \leq 0, \\ & -1 \leq x_i \leq 1, \end{aligned} \tag{4.4.1}$$

with starting point $\boldsymbol{x}^0 = (0.5, 0.5, \cdots, 0.5)^T$. The objective function $f_0(\boldsymbol{x})$ is strictly convex, but the nonlinear constraint functions $f_1(\boldsymbol{x})$, $f_2(\boldsymbol{x})$ are strictly concave. The numerical results are presented in Tables 4.2 and 4.3.

No performance increase is recorded for the LSQP routine. The execution time is almost exactly the same for all problem sizes as that of the double precision solver. However, the NAG E04NQF routine does show a decrease in execution time. Closer inspection reveals that the KKT (r^*) values for the LSQP-single routine in Table 4.2 are almost exactly the same as those for the LSQP-double routine, whereas the r^* values for E04NQFindicate that the problem has been solved to a much lower accuracy. The LSQP routine uses more subproblem evaluations in single precision, *e.g.* for a problem size of $n = 2560$ it uses 4396 in single and 3741 in double. The faster execution time of single precision calculations make up for the higher accuracy of the double precision calculations. The double precision solver for E04NQF however, requires 4× that of the single precision solver 1054905 in single precision versus 4997840 in double precision respectively. This explains the low accuracy of the single precision solution.

This single precision result, although not as accurate as the double precision result, produces a performance increase of 3.24×. This exceeds the 2× speed increase suggested by Buttari *et al.* (2008) for mixed precision algorithms. This increased performance is caused in part by the problem dependency of the QP solver. The more complex the subproblems that are solved in single precision, the higher the gains per subproblem evaluation. Furthermore speed is increased by performing single precision arithmetic because of the higher data rate. The improved performance for these problems is correlated to the amount of time spent on each subproblem evaluation by the QP single or double precision solver. The longer it takes the double precision solver to solve the subproblems, the larger the gain to be had with the single precision solver.

Note that the larger the problems become, the higher the computational complexity, which calls for higher precision to attain valid results. Therefore the performance decreases if the computational cost becomes too high, for example with problem sizes $n > 2560$. The E04NQF routine returns a warning on some iterations:

```
** Numerical difficulties have been encountered and no further progress
** can be made.
** ABNORMAL EXIT from NAG Library routine SP_E04: IFAIL =    10
** NAG soft failure - control returned
```

The error definition states that there have been numerical errors trying to satisfy the general constraints. The basis is very ill-conditioned. This error may be attributed to the low single precision accuracy. Relaxation could extend the range in which numerical errors affect the subproblems. However, this lies outside the scope of this study. Section 8.3 reflects more on this point.

4.4.1.2 Svanberg's second test problem

The second non-convex problem is formulated as

$$\begin{aligned} \min_{\boldsymbol{x}} \; f_0(\boldsymbol{x}) &= -\boldsymbol{x}^T \boldsymbol{S}\boldsymbol{x}, \\ \text{subject to } f_1(\boldsymbol{x}) &= \boldsymbol{x}^T \boldsymbol{P}\boldsymbol{x} - \frac{n}{2} \leq 0, \\ f_2(\boldsymbol{x}) &= \boldsymbol{x}^T \boldsymbol{Q}\boldsymbol{x} - \frac{n}{2} \leq 0, \\ -1 &\leq x_i \leq 1, \end{aligned} \tag{4.4.2}$$

with starting point $\boldsymbol{x}^0 = (0.25, 0.25, \cdots, 0.25)^T$. This time, the objective function $f_0(\boldsymbol{x})$ is strictly concave, while the nonlinear constraint functions $f_1(\boldsymbol{x})$, $f_2(\boldsymbol{x})$ are strictly convex. Numerical results for these problems are presented in Tables 4.4 and 4.5. Note that Svanberg (2002) used a different stopping criterion for both these test problems.

Once again, no performance increase is recorded for the `LSQP` routine. The same behaviour as in the first problem is reflected for the `E04NQF` subsolver. The performance increases, but the accuracy of the KKT conditions is less. This result for Svanberg's second test problem produces a performance increase of 2.46×.

Similarly, numerical difficulties are encountered for this problem when the number of design variables become too large. Figure 4.1 shows the performance difference for the single precision versus double precision case, whereas Figure 4.2 depicts the single precision versus mixed precision case.

4.4.2 Cam design problem

The cam design problem (Dolan *et al.*, 2004) aims to maximize the area of a valve opening for one rotation of a convex cam, with constraints on the curvature and on the radius of the cam.

Assume that the shape of the cam is circular over an angle of $6/5\pi$ of its circumference, with radius r_{min}. The design variables r_i, $i = 1, ..., n$, represent the radius of the cam at equally spaced angles distributed over an angle of $2/5\pi$. The area of the valve opening may be maximized by maximizing

$$f_0(\boldsymbol{r}) = \pi r_v^2 \left(\frac{1}{n} \sum_{i=1}^{n} r_i \right),$$

subject to constraints on the r_i. The design parameter r_v is related to the geometry of the valve. In addition it is required that $r_{min} \leq r_i \leq r_{max} \; \forall \; i$. The requirement that the cam be convex is expressed by requiring that

$$2r_{i-1}r_{i+1}\cos(\theta) - r_i(r_{i-1} + r_{i+1}) \leq 0, \qquad i = 0, \cdots, n+1,$$

where $r_{-1} = r_0 = r_{min}$, $r_{n+1} = r_{max}$, $r_{n+2} = r_n$, and $\theta = 2\pi/5(n+1)$. The curvature requirement is expressed by requiring

$$-\alpha - \left(\frac{r_{i+1} - r_i}{\theta} \right) \leq 0, \qquad i = 0, ..., n,$$

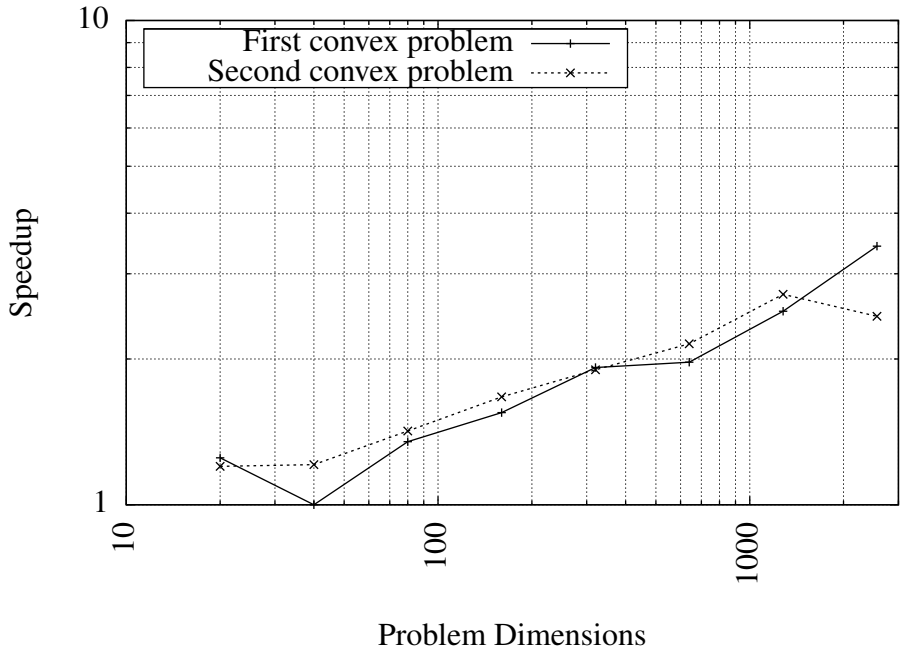


Figure 4.1: Svanberg's first and second test problems: speedups for the single precision over the double precision method.

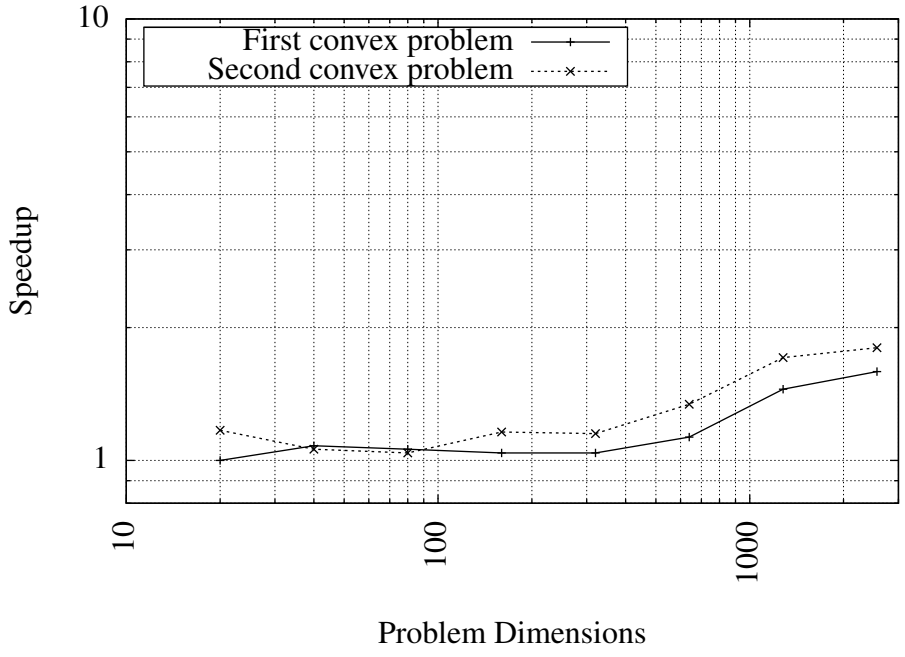


Figure 4.2: Svanberg's first and second test problems: speedups for the mixed precision over the double precision.

and

$$\left(\frac{r_{i+1} - r_i}{\theta}\right) - \alpha \leq 0, \qquad i = 0, ..., n.$$

Our formulation results in $3n + 4$ constraints; this is a slight departure from Dolan *et al.* (2004), where $2n$ constraint were formulated (our implementation requires that the constraints are written in negative-null form). The bounds $r_{min} = 1.0$ and $r_{max} = 2.0$ restrict the area of the valve, r_i, $r_v = 1.0$, and $\alpha = 1.5$ in the curvature constraint. Since the optimal cam shape is symmetric, one considers only half of the design angle.

The cam design problem produces very unfavourable results for single precision. The `LSQP` solver does well in solving the subproblems; however, there is still no performance increase and the performance fluctuates for different problem sizes, with the performance being better for $n = 1750$ and 2000 than for $n = 1500$. The `E04NQF` solver fails completely for this problem, with exit statuses: `IFAIL = 5`, `IFAIL = 6` and `IFAIL = 10`. All these failures have numerical errors in common.

`IFAIL = 5` states that the problem is infeasible. The problem is feasible in double precision, however constraints have been truncated to infeasible single precision constraints. Accordingly, `IFAIL = 6` states that the problem is infeasible or badly scaled. Lastly, `IFAIL = 10` states that there were numerical errors in trying to satisfy the general constraints and that the basis is very ill-conditioned.

Of particular interest here, however, is that the number of outer iterations of the `LSQP` solver are worse for this design problem. High numbers of outer iterations are undesirable for FEA (finite element analysis) problems, because many function evaluations occur at each update of the model. Once again the problem dependency of QP subsolvers is emphasized. It was necessary to enforce convergence for the `E04NQF` subsolver. The shape of the objective function causes the result to oscillate between two points.

4.4.3 Vanderplaats' cantilever beam

Consider the optimal sizing design of the tip-loaded multi-segmented cantilever beam proposed by Vanderplaats (2001). The beam is of fixed length l, is divided into p segments, and is subject to geometric, and stress constraints and a single displacement constraint. The geometry has been chosen such that a very large number of the constraints are active or 'near-active' at the optimum.

The objective function is formulated in terms of the design variables b_i and h_i as

$$\min \; f_0(\boldsymbol{b}, \boldsymbol{h}) = \sum_{i=1}^{k} b_i h_i l_i,$$

with l_i constant for given k. The bound constraints $1.0 \leq b_i \leq 80$ are enforced, and $5.0 \leq h_i \leq 80$ (the upper bounds were arbitrarily chosen; they are required in the dual formulation, and also needed to allow for the notion of a 'move limit'). The stress constraints are

$$\frac{\sigma(\boldsymbol{b}, \boldsymbol{h})}{\bar{\sigma}} - 1 \leq 0, \qquad i = 1, 2, p,$$

while the linear geometric constraints are written as

$$h_i - 20b_i \leq 0, \qquad i = 1, 2, p.$$

The constraints are rather easily written in terms of the design variables $\boldsymbol{b}$ and $\boldsymbol{h}$, see (Vanderplaats, 2001). Note that the constraints are normalized; this is sound practice in primal algorithms.

Using consistent units, the geometric and problem data are as follows (Etman *et al.*, 2009): a tip load of $P = 50000$, a modulus of elasticity $E = 2 \times 10^7$ and a beam length $l = 500$, while $\bar{\sigma} = 14000$ and $\bar{u} = 2.5$ are used. The starting point is $b_i = 5.0$ and $h_i = 60$ for all i, while $\epsilon_x = 10^{-5}$. The quadratic approximation to the reciprocal approximation is used for the objective and all the constraint values. (Since the objective is linear, even better results may be obtained by spherical quadratic approximations for the objective.) The problem is expressed in terms of $2p$ design variables, and $2p + 1$ constraints.

This problem does not see a speed increase for either of the single precision algorithms. According to the mixed precision routine, no acceleration in performance is delivered, although very little time is lost by updating the solution to double precision. Note here that the execution time does differ between QP subsolvers, once again emphasizing the problem dependency of QP subsolvers.

4.4.4 Svanberg's n-variate cantilever beam

Svanberg's n-variate cantilever beam is a cantilever problem proposed by Svanberg (1987) extended for solution with multiple design variables and constraints, and not only five design variables and constraints as in Svanberg (1987).

This problem delivers the best performance of all the problems investigated thus far. The reader is referred to Figure 4.3, which summarizes the performance results for the single versus double precision case, represented by the curves labelled S vs. D, and the mixed precision case, represented by the curves labelled M vs. D.

The sudden increase in performance at $n = 2000$ is real, although it is not easily explained. This problem was tested several times and it appears to be true. The performance does seem to stabilize, and later decreases when the problem size becomes too large. A maximum gain in speed of 29× is recorded. The data for the mixed precision results closely reflect those of the single precision results and are therefore highly favourable.

The data for this problem show that large accelerations are to be attained, although the speed increase is highly dependent on the problem and the problem size. Further investigation into the changing of subproblem evaluation limits shows that it may have profound affects on the performance of this particular problem. From Table 4.11 it can be seen that decreasing this limit for `E04NQF` decreases the simulation time, but increases the amount of outer iterations necessary for the problem to converge. Keeping in mind that this is an FE (finite element) problem, high numbers of outer iterations are undesirable, because many function evaluations occur at each update of the model.

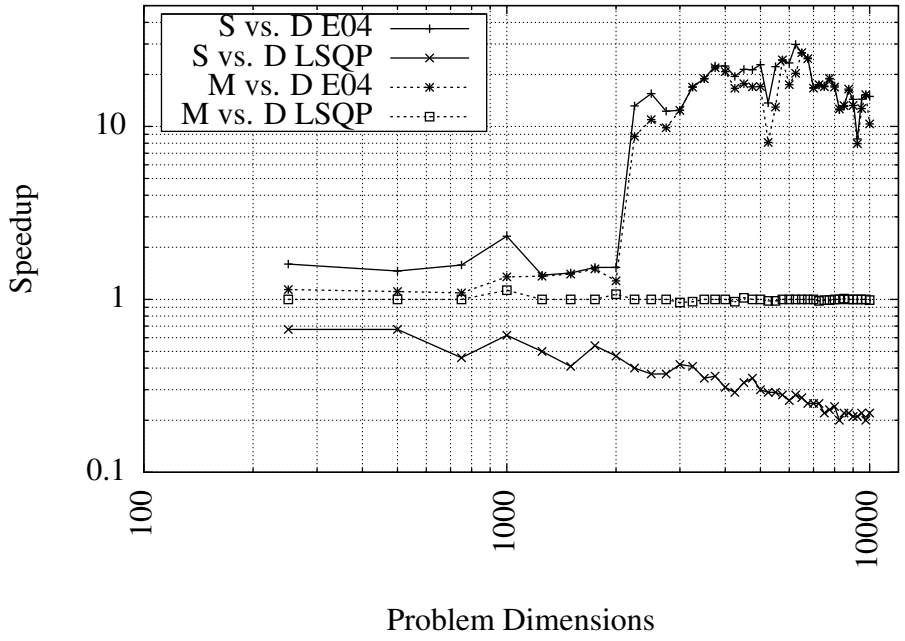


Figure 4.3: Five-variate cantilever problem: speedups for the mixed precision over the double precision.

4.5 Summary

It has been found that the performance of QP solvers in general, not only the performance between single and double precision, is highly problem dependent. This is shown by the large difference in performance between the `E04NQF` and the `LSQP` subsolvers for all test problems in this chapter. The tests show that great speed increases can be attained by performing some of the more computationally expensive subproblems, eg. in Sections 4.4.1.1, 4.4.1.2 and 4.4.4, in single precision for a specific subsolver. However, these accelerations are not only determined by whether one is using single or double precision, but also by the problem size. In the problems in Sections 4.4.1.1, 4.4.1.2 and 4.4.4, the speed increases for larger problem sizes and then drops down once the problem size becomes too large.

From the above one may conclude that it is beneficial to use single precision solvers for certain QP subsolvers and only for a certain problem size interval. In most cases, using a different QP subsolver may be more beneficial than changing to single precision.

In the following chapter, Chapter 5, a parallel environment is proposed in which one may exploit the problem dependency of QP and dual subsolvers. By solving the dual and QP subproblems in parallel and terminating the competing thread whenever a subsolver returns a viable solution, it is possible to always select the most efficient subsolver for solving the current approximation.

		Double precision					Single precision					
n	m	k^*	f_0^*	h^*	r^*	CPU	k^*	f_0^*	h^*	r^*	CPU	Speedup
					LSQP							
20	2	59	6.159	-3.82×10^{-09}	9.76×10^{-05}	0.04	59	6.159	-3.19×10^{-08}	9.76×10^{-05}	0.04	1.00
40	2	91	9.220	-6.36×10^{-09}	2.28×10^{-04}	0.10	93	9.220	4.91×10^{-08}	2.72×10^{-04}	0.11	0.91
80	2	117	19.671	-5.85×10^{-09}	2.69×10^{-04}	0.29	121	19.671	2.50×10^{-08}	5.21×10^{-04}	0.32	0.91
160	2	150	40.582	-5.62×10^{-09}	2.30×10^{-04}	1.03	139	40.582	-1.39×10^{-08}	1.64×10^{-03}	1.01	1.02
320	2	177	82.466	-1.29×10^{-08}	4.22×10^{-04}	4.05	165	82.466	-5.65×10^{-08}	4.56×10^{-04}	3.89	1.04
640	2	222	166.368	-1.18×10^{-08}	3.01×10^{-04}	18.28	216	166.368	1.01×10^{-08}	3.85×10^{-04}	18.19	1.00
1280	2	221	334.375	-1.44×10^{-08}	5.24×10^{-04}	69.11	226	334.375	1.16×10^{-08}	4.25×10^{-04}	72.12	0.96
2560	2	264	670.666	-1.40×10^{-08}	5.08×10^{-04}	321.17	269	670.666	8.61×10^{-08}	1.55×10^{-03}	333.36	0.96
					E04NQF							
20	2	59	6.159	-3.86×10^{-09}	9.77×10^{-05}	0.05	49	6.159	-2.75×10^{-07}	1.87×10^{-03}	0.04	1.25
40	2	92	9.220	-9.53×10^{-09}	2.00×10^{-04}	0.12	89	9.220	-7.61×10^{-08}	1.19×10^{-03}	0.12	1.00
80	2	120	19.671	-8.08×10^{-09}	2.80×10^{-04}	0.35	94	19.671	2.16×10^{-10}	4.47×10^{-03}	0.26	1.35
160	2	136	40.582	-1.17×10^{-08}	3.08×10^{-04}	1.30	100	40.582	-2.77×10^{-08}	6.87×10^{-03}	0.84	1.55
320	2	182	82.466	-7.27×10^{-09}	3.79×10^{-04}	7.09	107	82.469	-8.29×10^{-09}	3.10×10^{-02}	3.70	1.92
640	2	236	166.368	-7.05×10^{-09}	3.36×10^{-04}	44.66	149	166.373	-3.07×10^{-08}	3.57×10^{-02}	22.67	1.97
1280	2	231	334.375	-1.50×10^{-08}	4.53×10^{-04}	366.61	153	334.387	-7.17×10^{-10}	6.34×10^{-02}	146.33	2.51
2560	2	280	670.666	-4.30×10^{-09}	4.71×10^{-04}	3496.69	164	670.694	5.54×10^{-11}	1.22×10^{-01}	1022.79	3.42

Table 4.2: Svanberg's first test problem: comparison of the computational effort.

		Double precision					Mixed precision					
n	m	k^*	f_0^*	h^*	r^*	CPU	k^*	f_0^*	h^*	r^*	CPU	Speedup
20	2	59	6.159	-3.86×10^{-09}	9.77×10^{-05}	0.06	59	6.159	-3.97×10^{-09}	9.50×10^{-05}	0.06	1.00
40	2	92	9.220	-9.53×10^{-09}	2.00×10^{-04}	0.14	94	9.220	-2.22×10^{-09}	1.55×10^{-04}	0.13	1.08
80	2	120	19.671	-8.08×10^{-09}	2.80×10^{-04}	0.37	118	19.671	-5.53×10^{-09}	1.95×10^{-04}	0.35	1.06
160	2	136	40.582	-1.17×10^{-08}	3.08×10^{-04}	1.33	136	40.582	-7.92×10^{-09}	3.40×10^{-04}	1.28	1.04
320	2	182	82.466	-7.27×10^{-09}	3.79×10^{-04}	7.14	188	82.466	-8.40×10^{-09}	4.13×10^{-04}	6.86	1.04
640	2	236	166.368	-7.05×10^{-09}	3.36×10^{-04}	45.86	243	166.368	-6.05×10^{-09}	2.91×10^{-04}	40.64	1.13
1280	2	231	334.375	-1.50×10^{-08}	4.53×10^{-04}	364.73	228	334.375	-4.14×10^{-09}	4.83×10^{-04}	252.25	1.45
2560	2	280	670.666	-4.30×10^{-09}	4.71×10^{-04}	3301.13	289	670.666	-5.15×10^{-09}	4.74×10^{-04}	2072.26	1.59

Table 4.3: Svanberg's second test problem: speedup of mixed precision over double precision.

		Double precision					Single precision					
n	m	k^*	f_0^*	h^*	r^*	CPU	k^*	f_0^*	h^*	r^*	CPU	Speedup
					LSQP							
20	2	65	-13.841	4.76×10^{-09}	1.02×10^{-04}	0.04	65	-13.841	1.70×10^{-08}	1.03×10^{-04}	0.05	0.80
40	2	124	-30.780	1.32×10^{-09}	2.00×10^{-04}	0.14	123	-30.780	-8.77×10^{-08}	2.47×10^{-04}	0.15	0.93
80	2	161	-60.329	3.44×10^{-09}	3.92×10^{-04}	0.41	164	-60.329	4.36×10^{-07}	3.15×10^{-04}	0.44	0.93
160	2	213	-119.418	6.19×10^{-09}	4.01×10^{-04}	1.51	214	-119.418	7.32×10^{-08}	4.10×10^{-04}	1.57	0.96
320	2	284	-237.534	8.42×10^{-09}	4.78×10^{-04}	6.54	290	-237.534	1.01×10^{-06}	4.51×10^{-04}	6.87	0.95
640	2	354	-473.632	7.56×10^{-09}	4.31×10^{-04}	29.27	352	-473.632	2.02×10^{-06}	4.47×10^{-04}	29.84	0.98
1280	2	383	-945.625	4.13×10^{-09}	5.48×10^{-04}	119.96	398	-945.625	4.38×10^{-06}	4.28×10^{-04}	127.07	0.94
2560	2	459	-1889.334	2.44×10^{-09}	6.04×10^{-04}	557.66	463	-1889.334	5.60×10^{-07}	4.24×10^{-03}	573.97	0.97
					E04NQF							
20	2	65	-13.841	4.94×10^{-09}	1.02×10^{-04}	0.06	58	-13.841	6.00×10^{-07}	1.70×10^{-03}	0.05	1.20
40	2	124	-30.780	1.13×10^{-09}	2.02×10^{-04}	0.17	115	-30.780	8.94×10^{-07}	2.04×10^{-03}	0.14	1.21
80	2	160	-60.329	3.25×10^{-09}	4.19×10^{-04}	0.47	129	-60.329	5.17×10^{-09}	4.30×10^{-03}	0.33	1.42
160	2	214	-119.418	5.82×10^{-09}	4.06×10^{-04}	2.06	155	-119.418	6.08×10^{-09}	9.15×10^{-03}	1.23	1.67
320	2	284	-237.534	6.76×10^{-09}	5.32×10^{-04}	11.01	195	-237.531	8.72×10^{-08}	2.66×10^{-02}	5.80	1.90
640	2	356	-473.632	7.79×10^{-09}	4.63×10^{-04}	68.04	236	-473.624	4.29×10^{-08}	5.26×10^{-02}	31.65	2.15
1280	2	389	-945.625	4.51×10^{-09}	4.18×10^{-04}	599.36	284	-945.608	5.48×10^{-11}	8.77×10^{-02}	220.25	2.72
2560	2	440	-1889.334	4.32×10^{-09}	7.54×10^{-04}	5185.23	360	-1889.310	7.83×10^{-09}	1.05×10^{-01}	2118.57	2.45

Table 4.4: Svanberg's second test problem: comparison of the computational effort.

		Double precision					Mixed precision					
n	m	k^*	f_0^*	h^*	r^*	CPU	k^*	f_0^*	h^*	r^*	CPU	Speedup
20	2	65	-13.841	4.94×10^{-09}	1.02×10^{-04}	0.07	64	-13.841	6.02×10^{-09}	1.19×10^{-04}	0.06	1.17
40	2	124	-30.780	1.13×10^{-09}	2.02×10^{-04}	0.18	121	-30.780	3.52×10^{-09}	3.65×10^{-04}	0.17	1.06
80	2	160	-60.329	3.25×10^{-09}	4.19×10^{-04}	0.49	168	-60.329	3.14×10^{-09}	2.51×10^{-04}	0.47	1.04
160	2	214	-119.418	5.82×10^{-09}	4.06×10^{-04}	2.08	211	-119.418	7.20×10^{-09}	4.51×10^{-04}	1.79	1.16
320	2	284	-237.534	6.76×10^{-09}	5.32×10^{-04}	11.06	303	-237.534	7.08×10^{-09}	5.47×10^{-04}	9.62	1.15
640	2	356	-473.632	7.79×10^{-09}	4.63×10^{-04}	67.82	350	-473.632	4.01×10^{-09}	6.77×10^{-04}	50.61	1.34
1280	2	389	-945.625	4.51×10^{-09}	4.18×10^{-04}	599.64	421	-945.625	1.34×10^{-08}	5.90×10^{-04}	349.94	1.71
2560	2	440	-1889.334	4.32×10^{-09}	7.54×10^{-04}	5197.59	472	-1889.334	5.10×10^{-09}	7.04×10^{-04}	2880.09	1.80

Table 4.5: Svanberg's second test problem: speedup of mixed precision over double precision.

		Double precision					Single precision					
n	m	k^*	f_0^*	h^*	r^*	CPU	k^*	f_0^*	h^*	r^*	CPU	Speedup
					LSQP							
500	1504	14	-4.275	1.9×10^{-14}	2.1×10^{-08}	9.56	20	-4.274	2.9×10^{-08}	3.2×10^{-02}	8.38	0.84
750	2254	15	-4.274	7.6×10^{-14}	5.8×10^{-08}	18.10	24	-4.274	2.9×10^{-08}	8.7×10^{-03}	21.96	0.87
1000	3004	15	-4.273	1.3×10^{-13}	2.9×10^{-07}	28.12	22	-4.274	2.9×10^{-08}	2.2×10^{-02}	25.41	0.85
1250	3754	15	-4.273	3.9×10^{-13}	6.4×10^{-08}	36.16	63	-4.273	2.9×10^{-08}	2.9×10^{-02}	142.01	0.85
1500	4504	15	-4.273	2.0×10^{-13}	1.1×10^{-06}	45.89	109	-4.272	2.9×10^{-08}	2.9×10^{-02}	274.82	0.85
1750	5254	16	-4.273	9.1×10^{-13}	3.4×10^{-07}	68.19	28	-4.273	2.9×10^{-08}	2.9×10^{-02}	66.32	0.85
2000	6004	16	-4.273	6.3×10^{-13}	7.1×10^{-08}	97.14	23	-4.274	2.9×10^{-08}	3.1×10^{-02}	77.18	0.84
2250	6754	16	-4.273	5.7×10^{-13}	3.4×10^{-07}	99.16	42	-4.270	2.9×10^{-08}	4.3×10^{-02}	142.31	0.86
2500	7504	16	-4.273	9.0×10^{-13}	2.2×10^{-07}	110.07	42	-4.272	2.9×10^{-08}	5.0×10^{-02}	163.59	0.87
2750	8254	16	-4.273	8.1×10^{-13}	4.9×10^{-07}	116.49	23	-4.269	2.9×10^{-08}	5.5×10^{-02}	184.73	0.93
					E04NQF							
500	1504	9	-4.275	1.1×10^{-14}	3.3×10^{-09}	14.56						
750	2254	9	-4.274	2.5×10^{-14}	3.7×10^{-09}	26.37						
1000	3004	9	-4.273	3.5×10^{-14}	3.3×10^{-02}	45.98	** Numerical difficulties have been encountered					
1250	3754	9	-4.273	4.0×10^{-12}	2.8×10^{-02}	89.21	** and no further progress can be made.					
1500	4504	11	-4.273	5.8×10^{-12}	2.7×10^{-02}	136.08	** ABNORMAL EXIT from NAG Library routine					
1750	5254	9	-4.273	1.1×10^{-09}	1.8×10^{-02}	146.56	** SP_E04: IFAIL = 5,6 or 10					
2000	6004	9	-4.273	2.6×10^{-12}	2.5×10^{-02}	171.22	** NAG soft failure - control returned					
2250	6754	9	-4.273	3.8×10^{-12}	9.9×10^{-03}	251.76						
2500	7504	9	-4.273	1.7×10^{-12}	2.5×10^{-02}	287.35						
2750	8254	11	-4.273	1.4×10^{-12}	2.4×10^{-02}	373.96						

Table 4.6: Cam design problem: comparison of the computational effort.

		Double precision					Single precision					
n	m	k^*	f_0^*	h^*	r^*	CPU	k^*	f_0^*	h^*	r^*	CPU	Speedup
					LSQP							
2500	2500	10	53749	1.0×10^{-10}	3.1×10^{-06}	3.70	10	53749	4.2×10^{-07}	8.5×10^{-05}	4.42	0.84
3000	3000	10	53743	4.8×10^{-13}	1.8×10^{-07}	5.17	10	53743	4.0×10^{-07}	8.0×10^{-05}	5.92	0.87
3500	3500	10	53739	4.5×10^{-14}	8.9×10^{-08}	6.86	10	53739	4.3×10^{-07}	7.2×10^{-05}	8.08	0.85
4000	4000	11	53736	-9.9×10^{-16}	1.4×10^{-09}	10.14	11	53736	3.9×10^{-07}	6.4×10^{-05}	11.89	0.85
4500	4500	10	53733	1.1×10^{-09}	6.0×10^{-06}	11.10	10	53733	2.6×10^{-07}	6.6×10^{-05}	13.00	0.85
5000	5000	10	53731	1.0×10^{-10}	1.5×10^{-06}	13.64	10	53731	4.5×10^{-07}	6.4×10^{-05}	16.01	0.85
5500	5500	10	53730	7.8×10^{-12}	5.8×10^{-07}	16.50	10	53730	3.6×10^{-07}	6.1×10^{-05}	19.73	0.84
6000	6000	11	53728	-3.2×10^{-15}	9.5×10^{-10}	21.82	11	53728	4.0×10^{-07}	5.7×10^{-05}	25.26	0.86
6500	6500	11	53727	-3.9×10^{-15}	2.6×10^{-10}	27.70	11	53727	4.1×10^{-07}	5.3×10^{-05}	31.92	0.87
7000	7000	10	53726	5.3×10^{-10}	2.5×10^{-06}	31.71	10	53726	4.6×10^{-07}	5.1×10^{-05}	34.15	0.93
					E04NQF							
2500	2500	10	53749	1.0×10^{-10}	3.1×10^{-06}	7.10	10	53749	1.2×10^{-05}	3.8×10^{-05}	8.01	0.89
3000	3000	10	53743	7.4×10^{-13}	1.8×10^{-07}	10.53	10	53743	7.1×10^{-06}	4.0×10^{-05}	11.79	0.89
3500	3500	10	53739	6.5×10^{-12}	8.9×10^{-08}	14.92	10	53739	7.1×10^{-06}	3.1×10^{-05}	16.50	0.90
4000	4000	11	53736	7.3×10^{-13}	1.4×10^{-09}	21.39	11	53736	6.6×10^{-06}	2.5×10^{-05}	23.49	0.91
4500	4500	10	53733	1.1×10^{-09}	6.0×10^{-06}	25.60	10	53733	7.1×10^{-06}	2.7×10^{-05}	27.92	0.92
5000	5000	10	53731	1.0×10^{-10}	1.5×10^{-06}	30.94	10	53731	1.2×10^{-05}	2.8×10^{-05}	34.05	0.91
5500	5500	10	53730	8.2×10^{-12}	5.8×10^{-07}	37.26	10	53730	7.6×10^{-06}	3.0×10^{-05}	40.68	0.92
6000	6000	11	53728	7.6×10^{-13}	9.4×10^{-10}	47.24	11	53728	7.1×10^{-06}	2.5×10^{-05}	56.79	0.83
6500	6500	11	53727	7.3×10^{-13}	2.6×10^{-10}	55.70	11	53727	7.1×10^{-06}	2.5×10^{-05}	59.69	0.93
7000	7000	10	53726	5.3×10^{-10}	2.5×10^{-06}	60.00	10	53726	7.1×10^{-06}	2.3×10^{-05}	65.31	0.92

Table 4.7: Vanderplaats' cantilever beam: comparison of the computational effort.

		Double precision					Mixed precision					
n	m	k^*	f_0^*	h^*	r^*	CPU	k^*	f_0^*	h^*	r^*	CPU	Speedup
2500	2500	10	53749	1.0×10^{-10}	3.1×10^{-06}	7.32	10	53749	1.2×10^{-05}	3.8×10^{-05}	8.16	0.90
3000	3000	10	53743	7.4×10^{-13}	1.8×10^{-07}	10.84	10	53743	7.1×10^{-06}	4.0×10^{-05}	11.73	0.92
3500	3500	10	53739	6.5×10^{-12}	8.9×10^{-08}	15.31	10	53739	7.1×10^{-06}	3.1×10^{-05}	16.33	0.94
4000	4000	11	53736	7.3×10^{-13}	1.4×10^{-09}	21.73	11	53736	2.2×10^{-13}	7.2×10^{-05}	22.90	0.95
4500	4500	10	53733	1.1×10^{-09}	6.0×10^{-06}	26.09	10	53733	7.1×10^{-06}	2.7×10^{-05}	27.76	0.94
5000	5000	10	53731	1.0×10^{-10}	1.5×10^{-06}	31.65	10	53731	1.2×10^{-05}	2.8×10^{-05}	33.79	0.94
5500	5500	10	53730	8.2×10^{-12}	5.8×10^{-07}	38.43	10	53730	7.6×10^{-06}	3.0×10^{-05}	40.15	0.96
6000	6000	11	53728	7.6×10^{-13}	9.4×10^{-10}	48.32	11	53728	1.6×10^{-12}	6.1×10^{-05}	55.52	0.87
6500	6500	11	53727	7.3×10^{-13}	2.6×10^{-10}	56.41	11	53727	5.5×10^{-13}	5.7×10^{-05}	59.67	0.95
7000	7000	10	53726	5.3×10^{-10}	2.5×10^{-06}	62.15	10	53726	7.1×10^{-06}	2.3×10^{-05}	66.15	0.94

Table 4.8: Vanderplaats' cantilever beam: speedup of mixed precision over double precision.

		Double precision					Single precision					
n	m	k^*	f_0^*	h^*	r^*	CPU	k^*	f_0^*	h^*	r^*	CPU	Speedup
					LSQP							
250	1	7	1.310	1.33×10^{-11}	1.06×10^{-11}	0.02	7	1.310	-1.18×10^{-08}	5.20×10^{-09}	0.03	0.67
500	1	8	1.310	-4.73×10^{-13}	1.53×10^{-10}	0.04	8	1.310	-2.09×10^{-08}	2.13×10^{-08}	0.06	0.67
750	1	8	1.310	-5.12×10^{-13}	7.53×10^{-10}	0.06	8	1.310	1.69×10^{-08}	1.90×10^{-09}	0.13	0.46
1000	1	8	1.310	2.02×10^{-13}	2.34×10^{-09}	0.08	8	1.310	5.60×10^{-09}	2.42×10^{-09}	0.13	0.62
1250	1	8	1.310	1.93×10^{-13}	5.49×10^{-11}	0.10	8	1.310	-1.73×10^{-08}	1.69×10^{-09}	0.20	0.50
1500	1	8	1.310	-4.57×10^{-14}	1.10×10^{-10}	0.12	8	1.310	-1.16×10^{-08}	2.01×10^{-09}	0.29	0.41
1750	1	8	1.310	7.76×10^{-12}	1.63×10^{-10}	0.14	8	1.310	2.90×10^{-08}	2.79×10^{-08}	0.26	0.54
2000	1	9	1.310	-9.16×10^{-14}	3.12×10^{-10}	0.17	9	1.310	7.26×10^{-09}	1.64×10^{-09}	0.36	0.47
2250	1	9	1.310	-1.01×10^{-13}	4.76×10^{-10}	0.20	9	1.310	8.95×10^{-09}	1.44×10^{-09}	0.50	0.40
2500	1	9	1.310	-1.13×10^{-13}	6.94×10^{-10}	0.22	9	1.310	1.60×10^{-08}	1.41×10^{-09}	0.60	0.37
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
9000	1	10	1.310	-4.82×10^{-13}	7.54×10^{-08}	0.90	10	1.310	-9.06×10^{-09}	1.75×10^{-09}	4.25	0.21
9250	1	10	1.310	-4.95×10^{-13}	8.46×10^{-08}	0.92	10	1.310	-2.06×10^{-08}	7.49×10^{-09}	4.29	0.21
9500	1	10	1.310	-5.18×10^{-13}	9.44×10^{-08}	0.95	10	1.310	-2.37×10^{-10}	6.48×10^{-09}	4.38	0.22
9750	1	10	1.310	-5.12×10^{-13}	1.05×10^{-07}	0.98	10	1.310	-4.01×10^{-09}	3.84×10^{-09}	4.87	0.20
10000	1	10	1.310	-5.67×10^{-13}	1.16×10^{-07}	1.01	10	1.310	2.09×10^{-08}	4.62×10^{-09}	4.67	0.22
					E04NQF							
250	1	8	1.310	-2.22×10^{-16}	1.43×10^{-05}	0.08	8	1.310	1.47×10^{-11}	2.05×10^{-03}	0.05	1.60
500	1	8	1.310	5.04×10^{-14}	2.01×10^{-05}	0.41	10	1.310	1.18×10^{-10}	2.64×10^{-03}	0.28	1.46
750	1	8	1.310	2.76×10^{-13}	1.84×10^{-05}	1.45	11	1.310	-1.32×10^{-10}	3.95×10^{-03}	0.92	1.58
1000	1	8	1.310	1.23×10^{-12}	2.17×10^{-05}	4.20	11	1.310	7.54×10^{-11}	3.48×10^{-03}	1.81	2.32
1250	1	9	1.310	5.33×10^{-15}	2.29×10^{-05}	7.34	11	1.310	7.11×10^{-11}	7.44×10^{-03}	5.32	1.38
1500	1	9	1.310	4.22×10^{-15}	2.31×10^{-05}	12.81	12	1.310	3.95×10^{-11}	9.49×10^{-03}	9.04	1.42
1750	1	9	1.310	7.77×10^{-15}	2.85×10^{-05}	21.30	12	1.310	4.80×10^{-11}	7.33×10^{-03}	13.94	1.53
2000	1	9	1.310	1.35×10^{-14}	3.51×10^{-05}	31.55	13	1.310	3.82×10^{-11}	7.76×10^{-03}	20.59	1.53
2250	1	9	1.310	1.19×10^{-13}	2.52×10^{-05}	266.70	13	1.310	1.44×10^{-11}	8.16×10^{-03}	20.26	13.16
2500	1	9	1.310	1.71×10^{-13}	3.54×10^{-05}	319.53	13	1.310	2.22×10^{-11}	8.03×10^{-03}	20.64	15.48
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
9000	1	10	1.310	4.44×10^{-16}	6.73×10^{-05}	3746.36	16	1.311	1.16×10^{-11}	1.76×10^{-02}	261.34	14.34
9250	1	10	1.310	-9.99×10^{-16}	7.96×10^{-05}	2323.06	16	1.311	3.87×10^{-12}	1.74×10^{-02}	274.44	8.46
9500	1	10	1.310	-1.89×10^{-15}	9.55×10^{-05}	4139.68	16	1.311	1.80×10^{-11}	1.71×10^{-02}	288.01	14.37
9750	1	10	1.310	4.44×10^{-16}	8.85×10^{-05}	4533.11	16	1.310	3.42×10^{-11}	1.69×10^{-02}	297.96	15.21
10000	1	10	1.310	7.99×10^{-15}	7.96×10^{-05}	4796.70	16	1.311	1.24×10^{-10}	1.67×10^{-02}	321.61	14.91

Table 4.9: Svanberg's n-variate cantilever beam: comparison of the computational effort.

		Double precision				Mixed precision				
n	m	k^*	f_0^*	h^*	r^*	CPU	k^*	f_0^*	h^*	r^*	CPU	Speedup
250	1	8	1.310	-2.22×10^{-16}	1.43×10^{-05}	0.08	10	1.310	4.70×10^{-13}	9.64×10^{-06}	0.07	1.14
500	1	8	1.310	5.04×10^{-14}	2.01×10^{-05}	0.41	13	1.310	-1.11×10^{-16}	1.80×10^{-05}	0.37	1.11
750	1	8	1.310	2.76×10^{-13}	1.84×10^{-05}	1.45	14	1.310	1.09×10^{-14}	1.70×10^{-05}	1.33	1.09
1000	1	8	1.310	1.23×10^{-12}	2.17×10^{-05}	4.17	14	1.310	1.33×10^{-14}	1.29×10^{-05}	3.08	1.35
1250	1	9	1.310	5.33×10^{-15}	2.29×10^{-05}	7.30	11	1.310	7.11×10^{-11}	7.44×10^{-03}	5.35	1.36
1500	1	9	1.310	4.22×10^{-15}	2.31×10^{-05}	12.81	16	1.310	-4.44×10^{-16}	2.97×10^{-05}	9.18	1.40
1750	1	9	1.310	7.77×10^{-15}	2.85×10^{-05}	20.99	12	1.310	4.80×10^{-11}	7.33×10^{-03}	13.98	1.50
2000	1	9	1.310	1.35×10^{-14}	3.51×10^{-05}	31.14	17	1.310	-1.33×10^{-15}	3.66×10^{-05}	24.37	1.28
2250	1	9	1.310	1.19×10^{-13}	2.52×10^{-05}	266.76	17	1.310	1.78×10^{-15}	3.16×10^{-05}	30.48	8.75
2500	1	9	1.310	1.71×10^{-13}	3.54×10^{-05}	317.74	17	1.310	4.44×10^{-16}	2.28×10^{-05}	29.00	10.96
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
9000	1	10	1.310	4.44×10^{-16}	6.73×10^{-05}	3764.70	21	1.310	1.78×10^{-15}	4.45×10^{-05}	284.63	13.23
9250	1	10	1.310	-9.99×10^{-16}	7.96×10^{-05}	2297.84	21	1.310	5.33×10^{-15}	7.69×10^{-05}	289.11	7.95
9500	1	10	1.310	-1.89×10^{-15}	9.55×10^{-05}	4142.53	22	1.310	-1.22×10^{-15}	5.07×10^{-05}	326.50	12.69
9750	1	10	1.310	4.44×10^{-16}	8.85×10^{-05}	4514.44	23	1.310	1.33×10^{-15}	5.77×10^{-05}	295.93	15.26
10000	1	10	1.310	7.99×10^{-15}	7.96×10^{-05}	4760.98	22	1.310	4.00×10^{-15}	4.89×10^{-05}	460.52	10.34

Table 4.10: Svanberg's n-variate cantilever beam: speedup of mixed precision over double precision.

		Double precision					Mixed precision					
itlim	n	k^*	f_0^*	h^*	r^*	CPU	k^*	f_0^*	h^*	r^*	CPU	Speedup
250	1000	16	1.3103	1.9×10^{-15}	1.3×10^{-05}	0.32	16	1.3107	1.3×10^{-10}	4.6×10^{-03}	0.21	1.52
500	1000	9	1.3103	1.1×10^{-14}	2.5×10^{-05}	0.54	12	1.3106	1.5×10^{-10}	2.7×10^{-03}	0.46	1.17
750	1000	8	1.3103	1.3×10^{-12}	1.8×10^{-05}	1.39	12	1.3104	-9.8×10^{-12}	6.8×10^{-03}	1.15	1.21
1000	1000	8	1.3103	1.3×10^{-12}	2.6×10^{-05}	3.36	11	1.3104	3.5×10^{-11}	3.5×10^{-03}	1.79	1.88
1250	1000	8	1.3103	1.3×10^{-12}	2.9×10^{-05}	4.25	11	1.3104	7.5×10^{-11}	3.4×10^{-03}	1.87	2.27
1500	1000	8	1.3103	1.2×10^{-12}	2.1×10^{-05}	3.87	11	1.3104	7.5×10^{-11}	3.4×10^{-03}	1.86	2.08
1750	1000	8	1.3103	1.2×10^{-12}	2.1×10^{-05}	4.19	11	1.3104	7.5×10^{-11}	3.4×10^{-03}	1.84	2.28
2000	1000	8	1.3103	1.2×10^{-12}	2.1×10^{-05}	4.17	11	1.3104	7.5×10^{-11}	3.4×10^{-03}	1.83	2.28
2250	1000	8	1.3103	1.2×10^{-12}	2.1×10^{-05}	4.22	11	1.3104	7.5×10^{-11}	3.4×10^{-03}	1.89	2.23
2500	1000	8	1.3103	1.2×10^{-12}	2.1×10^{-05}	4.21	11	1.3104	7.5×10^{-11}	3.4×10^{-03}	1.83	2.30
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
4000	1000	11	1.3103	1.2×10^{-12}	2.1×10^{-05}	4.18	11	1.3104	7.5×10^{-11}	3.4×10^{-03}	1.82	2.30
4250	1000	11	1.3103	1.2×10^{-12}	2.1×10^{-05}	4.19	11	1.3104	7.5×10^{-11}	3.4×10^{-03}	1.83	2.29
4500	1000	11	1.3103	1.2×10^{-12}	2.1×10^{-05}	4.19	11	1.3104	7.5×10^{-11}	3.4×10^{-03}	1.83	2.29
4750	1000	11	1.3103	1.2×10^{-12}	2.1×10^{-05}	4.17	11	1.3104	7.5×10^{-11}	3.4×10^{-03}	1.83	2.28
5000	1000	11	1.3103	1.2×10^{-12}	2.1×10^{-05}	4.18	11	1.3104	7.5×10^{-11}	3.4×10^{-03}	1.83	2.28

Table 4.11: Svanberg's n-variate cantilever beam: computational effort with changing limits on the subproblem evaluations.

Chapter 5

Parallel dual and QP solvers

This chapter begins with a brief background on dual and QP methods in SAO, as well as thread cancellation. The implementation is described in light of the OpenMP 3.0 standard. The OpenMP standard allows the programmer to exit irregular parallel algorithms by setting appropriate flags. The benefits and disadvantages of using flagged thread cancellation are discussed. The implementation and the results for selected test problems are presented in Sections 5.3 and 5.4 respectively. The chapter concludes with a summary of the results.

5.1 Background

In the previous chapter it was found that the performance of QP solvers varies from one problem to the next. The same characteristics are displayed when solving problems using various QP and dual solvers. In general, the problem-specific properties, which influence performance, are not known in advance and it is impossible to judge which solver would be best suited for the problem at hand. In this chapter multiple search procedures are performed, in parallel, on a set of convex separable approximation functions.

By taking advantage of the convex QP method of the approximate subproblems proposed by Etman *et al.* (2009), it is possible to solve the functions in both dual and QP subspaces. Etman *et al.* (2009) found that QP methods are very promising alternatives in SAO programs when solving structural optimization problems for which both the number of design variables and the number of constraints are very high. On the other hand, dual methods remain the most efficient for problems with very many design variables and only a few constraints. Thus, by performing these two (or more) evaluations in parallel and terminating the competing thread whenever a viable solution is returned, it may be ensured that the most effective solver is used for the solution of the subproblem at each approximation.

The structural design problems defined in Sections 4.4.1 through 4.4.4 of Chapter 4 will be evaluated. These problems are chosen intentionally to take advantage of their individual properties. The cam design problem has $3n + 4$ constraints, where n is the number of design variables, therefore it is expected of the QP solvers to perform better with this problem. Accordingly, the dual solvers should perform better with Svanbergs' n-variate cantilever beam problem with

n design variables, and one displacement constraint. The last problem is chosen because it has $n+1$ constraints. Nearly balancing the number of design variables and constraints should display that dual and QP solvers can deliver similar performance on these types of problems. The dual `l-BFGS-b` and the QP `E04NQF` and `LSQP` solvers are used to solve the subproblems in parallel.

In order to evaluate the subproblems in parallel, the fork-join model for a shared memory system is used (Chandra *et al.*, 2001). This basic fork-join parallel programming concept is depicted in Figure 3.4. This approach to solving the subproblems creates a highly irregular parallel algorithm, because it contains subcomputations whose amount of work is not known in advance. Since the OpenMP 3.0 standard (OpenMP 3.0) does not support language-level thread cancellation. Flagged thread cancellation, proposed by Süß and Leopold (2006*b*), is used. This approach is highly dependent on the ability to cancel threads in a parallel region, which, according to Mattson (2003), one of the initial designers of the OpenMP specification, OpenMP was never designed to do. Süß and Leopold (2006*b*) provide a workaround that is implemented here. It will be shown how the lack of proper thread cancellation negatively affects the performance of the algorithm.

Parallel threads and the use of flags for inter-thread communication in a multithreaded environment are discussed in Section 5.2. This discussion is followed by a discussion of how threads are cancelled in irregular parallel algorithms and finally, the drawbacks of using flags as a workaround for language level thread cancellation are provided in Section 5.2.3. In Section 5.3 a parallel algorithm is proposed for flagged cancellation in the SAO environment. The results are evaluated and summarized for both design problems in Sections 5.4 and 5.5 respectively.

5.2 Threads and flags

Since it is not possible to use language-level thread cancellation within OpenMP, flags are used to indicate when execution must halt. The effect of flags could be detrimental to performance. If flags are iteratively checked in competing threads, and one of the competing threads is having difficulty during some iteration, the remaining set of threads must wait for a checkpoint to be reached in that thread, even if all the other competing threads have completed. If the information in the troublesome thread is not needed for further execution (as in our case), thread cancellation would be more effective than flagged cancellation.

Even though irregular parallel algorithms have rarely been written using OpenMP, it is possible to do so. Süß and Leopold (2006*b*) used a simple breadth-first search application as an example to show the deficiency of the OpenMP specification, which is the difficulty to cancel the threads within a parallel region. Süß and Leopold (2006*b*) outline a simple way to work around this issue within the existing OpenMP specification (OpenMP 3.0). However, their main contribution is a proposal for an extension of OpenMP with thread cancellation support. Thread cancellation, incorporating the use of flags, is used because the OMPi compiler developed by Nikolopoulos *et al.* (2001), which is also used by Süß and Leopold (2006*b*), is a lightweight open source OpenMP compiler and runtime system that was only developed for codes written in C and would need to be adapted to compile Fortran codes.

5.2.1 Irregular parallel algorithms

Irregular parallel algorithms are defined as subcomputations whose amount of work is not known in advance and hence the work can only be distributed at runtime. Important subclasses of irregular algorithms include algorithms using *taskpools* and *speculative algorithms* (Guerraoui, 2010). The focus is on the second type, where multiple algorithms are started in parallel, and it is then speculated which algorithm will exit first. If the exit conditions are satisfied by any thread, the competing threads are cancelled.

Other examples for irregular algorithms are search and sorting algorithms, graph algorithms, and more involved applications like volume rendering (Süß and Leopold, 2006*b*). According to Mattson (2003), one of the initial designers of the OpenMP specification, OpenMP was never meant for irregular applications. Authors such as Hisley *et al.* (2000), Dedu *et al.* (2000) and Nikolopoulos *et al.* (2001) have tried to use OpenMP for similar applications and have obtained mixed results.

5.2.2 Thread cancellation using flags

A general sketch of the parallel use of flags for thread cancellation is presented in pseudocode form in Algorithm 1, see Appendix E.2 for the complete source. The algorithm starts by assigning the maximum number of threads (line 1) to start inside the parallel region. This may be done anywhere in the code before the parallel region where the serial code is performed. Afterwards, a parallel region (line 3) is spawned. Then the thread numbers are set using the `omp_get_thread_num` function in order to keep track of the threads (this is not a necessity). Following the barrier on line 1, the parallel sections are started using the SECTIONS directive (line 1). Each SECTION (lines 1, 1 and 1) nested within the SECTIONS directive is executed once by each thread in the team.

Contained within each parallel task there is an iterative check, shown in Algorithm 2. Once either task has completed, a flag is set with the appropriate task number (lines 1 and 1). The iterative check will pick up that the value of the flag has changed and will immediately return from the task. If there has been no change in the flag value, the task continues until itself returns. Since OpenMP does not include forceful thread cancellation, one must wait until all threads have attained the changed flag state and have exited successfully. This becomes a big problem when the size of each task iteration is highly irregular, as is often the case with different QP solvers, as seen in Chapter 4.

There is no other known way in OpenMP to indicate that threads, within a parallel region, should end their work. The problems with this approach (Süß and Leopold, 2006*b*) are pointed out in Section 5.2.3. Section 5.2.4 lists three methods of thread cancellation.

5.2.3 The problem with flags

Great care needs to be taken by the programmer to ensure that all flags are checked correctly. Human error could be encountered when using flags to indicate that the parallel region should be aborted.

Algorithm 1: Parallel use of flags for thread cancellation.

```
1  omp_set_thread_num([maximum number of threads])
2  !$ omp parallel [clause ...]
3      tid = omp_get_thread_num()
4      !$ omp barrier
5      !$ omp sections
6          !$ omp section
7              parallel_Task(Task one arguments...)
8              if flag.le.0 then
9                  flag = 1
10             end
11             !$ omp barrier
12             !$ omp flush (tid,...)
13         !$ omp section
14             parallel_Task(Task two arguments...)
15             if flag.le.0 then
16                 flag = 2
17             end
18             !$ omp barrier
19             !$ omp flush (tid,...)
20         !$ omp section
               ⋮
21
22     !$ omp end sections
23 !$ omp end parallel
```

Algorithm 2: Return to end parallel task.

```
1  while Solution not found do
       ⋮
2
3      if flag.le.0 then
4          return
5      end
       ⋮
6
7  end
```

In Algorithm 1, a return might be found by any of the active tasks, but the algorithm then hangs in the implicit barrier at the end of each section (lines 1 and 1), because another thread has also returned with an exit. In this situation, the program will most likely exhibit deadlock[1]. In OpenMP, the sequence of barrier constructs must be the same for every thread in the team in order to exit correctly.

Therefore, the code in Algorithm 1 is not safe. However, the use of flags has yet another problem. According to the OpenMP memory model, flags must be manually updated with the use of a FLUSH directive before their values are guaranteed to be correct and up to date. Inexperienced OpenMP programmers often miss this step (Süß and Leopold, 2006*a*). The consequence is similar. The program will potentially deadlock because one flag will carry the old value and will not exit its respective section. To summarize:

1. there is no easy way to branch out of a parallel region in OpenMP; the use of flags is the only workaround;
2. working with flags becomes more difficult once barriers come into play;
3. deadlock may arise if the programmer forgets to flush a flag.

While these points emphasized by Süß and Leopold (2006*b*) warn the programmer, flagged cancellation as a viable workaround for the main problem is still cumbersome and error-prone. The proposal by Süß and Leopold (2006*b*) to incorporate language-level thread cancellation would be extremely useful for our scenario, where it is necessary to exit a speculative algorithm. The OMP*i* compiler developed by Nikolopoulos *et al.* (2001) provides an OpenMP extension to forcefully cancel flags. However, this compiler is only written for the C language and due to time constraints the code could not be used. In the following section, three types of language-level thread cancellation are summarized briefly.

5.2.4 Language-level thread cancellation

Language-level thread cancellation can be classified into three subclasses (Süß and Leopold, 2006*b*). Firstly, *forceful cancellation*, where a thread has the ability to cancel another thread from the outside. The cancelled thread may get the opportunity to clean up after itself, yet it does not have the power to decide when to be cancelled, nor to prevent cancellation at all. Asynchronous cancellation in POSIX threads is an example of forceful cancellation.

Another type of language-level thread cancellation is *deferred cancellation*, where the cancelled thread is not terminated immediately, but only at predefined cancellation points. Deferred cancellation is supported in POSIX threads.

Lastly, *cooperative cancellation*, by contrast, is when a thread can only request the cancellation of another thread. The cancelled thread has the opportunity to honor this request and cancel itself or process the request at a later time, and can even choose to ignore it altogether. Java threads support cooperative cancellation.

[1] A *deadlock* is a runtime situation that occurs when a thread is waiting for a resource that is never going to be available (Chapman *et al.*, 2008). It is sometimes referred to as *software lock* or *soft lock*.

Of most import is forceful cancellation, therefore a more in-depth look at Java threads will not be undertaken. However, forceful cancellation with the use of POSIX threads will also not be applied because of the platform dependencies discussed in Section 3.4.3.

5.3 Parallel dual and QP methods

Three subsolvers are used in parallel: two are based on quadratic approximations `LSQP` and `E04NQF`, and one is based on a quadratic dual approximation `l-BFGS-b`. Algorithm 3 displays the algorithm that has been applied to solve a test problem with all three solvers at once.

Algorithm 3: Parallel use of flags for thread cancellation as applied in SAO.

```
1  omp_set_thread_num(3)
2  !$ omp parallel default(shared)
3      !$ omp+private (tid)
4      tid = omp_get_thread_num()
5      !$ omp sections
6          !$ omp section
7              call drive_lbfgsb(n,x,...,problem specifics,...)
8              if (flag .le.0) then
9                  if (xkkt.le.kkt_tol.or.msg.eq.0...) then
10                     flag = 1
11                 end
12             end
13             !$ omp flush (tid,flag,problem specifics)
14         !$ omp section
15             call drive_e04nqf(n,x,...,problem specifics,...)
16             if (flag.le.0) then
17                 if (xkkt.le.kkt_tol.or.msg.eq.0...) then
18                     flag = 2
19                 end
20             end
21             !$ omp flush (tid,flag,problem specifics)
22         !$ omp section
23             call drive_lsqp(n,x,...,problem specifics,...)
24             if (flag.le.0) then
25                 if (xkkt.le.kkt_tol.or.msg.eq.0...) then
26                     flag = 3
27                 end
28             end
29             !$ omp flush (tid,flag,problem specifics)
30     !$ omp end sections
31 !$ omp end parallel
```

The number of threads to be started (line 3) is set equal to three because three different solvers are used. This could be any amount depending on the available solvers. From Algorithm 3 it is clearly seen how the three solvers are started in parallel. A flag may then only be set after one routine has been completed. Note that the KKT conditions are tested for on the subproblem level of each of the associated problems, as well as on any error conditions that might have been encountered during the solution of the subproblem.

Non-fatal errors are allowed for the subsolvers; for more information the reader is referred to the respective Warnings and Errors sections in the manuals. For the `l-BFGS-b` routine, only the error-(`msg`) value of 0 is allowed, for `E04NQF` `msg` $= 0, 5$ and for `LSQP` `msg` $= 0, -5$. The 5 and -5 errors are non-fatal, but indicate the presence of some numerical difficulties and 0 indicates that no errors were encountered. Therefore, it is possible to restrict all warnings and errors, but in some cases this causes the execution to be much slower, whereas increasing convexity in an inner iteration (see Equation (2.2.4)) could give better performance.

For the sake of brevity it has not been shown that the problem-specific values that must be updated are flushed for each solver. However, this is important to ensure data conformity. In addition, restating Algorithm 2 is omitted, as it is used in exactly the same manner as mentioned in Section 5.2.2 to return from the solution of the subproblem at some checkpoint.

5.4 Results

Three design problems are used to illustrate how solving the subproblems in parallel, and selecting the first correct result to return, compares with solving the problems with respective solvers in series. The cam design problem has $3n + 4$ constraints, where n is the number of design variables. Therefore it is expected that the QP solver will perform better with this problem. Accordingly, it is expected that the dual solver will perform better with Svanberg's n-variate cantilever beam problem with n design variables and one displacement constraint. The last problem is Vanderplaats' cantilever beam, as it has n design variables and $n + 1$ constraints.

In the tables, n indicates the number of design variables m, the number of constraints, k^* indicates the number of outer iterations, and l^* the number of inner iterations. The optimal function values are indicated with f^*, where h^* and r^* respectively indicate the maximum constraint violation and KKT-condition of the result. CPU is the real execution time and CPU_d is the execution time that would have been obtained if thread cancellation has been applied.

5.4.1 Cam design problem

The cam design problem (Dolan *et al.*, 2004) aims to maximize the area of a valve opening for one rotation of a convex cam, with constraints on the curvature and on the radius of the cam. This problem is discussed in Section 4.4.2.

The most apparent observation is the difference in the performance between the dual and QP solvers. The performance gain is more than an order of magnitude, and for problems where n is large it becomes too difficult to solve with the dual solver. On the other hand, the QP solver still

Table 5.1: Serial execution of cam design problem–`l-BFGS-b`.

n	m	k^*	l^*	f^*	h^*	r^*	CPU
2	10	6	49	-4.785	1.37×10^{-05}	3.33×10^{-04}	0.05
4	16	6	31	-4.542	1.18×10^{-06}	1.56×10^{-04}	0.28
6	22	9	72	-4.451	5.45×10^{-05}	6.15×10^{-04}	2.91

Table 5.2: Serial execution of cam design problem–`E04NQF`.

n	m	k^*	l^*	f^*	h^*	r^*	CPU
250	754	9	0	-4.277	8.88×10^{-16}	3.04×10^{-08}	0.59
500	1504	9	0	-4.275	1.51×10^{-14}	3.38×10^{-09}	2.96
750	2254	9	0	-4.274	2.49×10^{-14}	3.74×10^{-09}	6.25
1000	3004	11	5	-4.274	9.93×10^{-12}	3.28×10^{-02}	17.68
1250	3754	9	0	-4.274	4.07×10^{-12}	2.85×10^{-02}	16.76
1500	4504	23	28	-4.274	5.90×10^{-12}	3.08×10^{-02}	119.06
1750	5254	13	5	-4.274	3.18×10^{-12}	2.68×10^{-02}	62.82
2000	6004	22	4	-4.273	2.18×10^{-12}	2.19×10^{-02}	112.47
2250	6754	38	42	-4.273	2.14×10^{-12}	2.64×10^{-02}	416.71
2500	7504	20	13	-4.273	1.79×10^{-12}	3.21×10^{-02}	224.90

Table 5.3: Serial execution of cam design problem–`LSQP`.

n	m	k^*	l^*	f^*	h^*	r^*	CPU
250	754	13	0	-4.277	3.36×10^{-13}	2.07×10^{-08}	3.60
500	1504	14	0	-4.275	1.91×10^{-14}	2.17×10^{-08}	9.92
750	2254	15	0	-4.274	7.64×10^{-14}	5.87×10^{-08}	19.11
1000	3004	15	0	-4.274	1.37×10^{-13}	3.00×10^{-07}	29.78
1250	3754	15	0	-4.274	3.96×10^{-13}	6.42×10^{-08}	38.78
1500	4504	15	0	-4.274	2.01×10^{-13}	1.14×10^{-06}	49.30
1750	5254	16	1	-4.274	3.94×10^{-13}	1.50×10^{-07}	77.21
2000	6004	16	0	-4.273	6.31×10^{-13}	7.18×10^{-08}	103.89
2250	6754	16	0	-4.273	5.79×10^{-13}	3.46×10^{-07}	109.45
2500	7504	16	0	-4.273	9.03×10^{-13}	2.27×10^{-07}	119.08

Table 5.4: Parallel execution of cam design problem.

n	m	k^*	l^*	f^*	h^*	r^*	CPU	CPU_d
250	754	9	0	-4.277	8.88×10^{-16}	3.04×10^{-08}	6.06	1.89
500	1504	9	0	-4.275	1.91×10^{-14}	9.63×10^{-09}	15.33	8.35
750	2254	9	0	-4.274	7.37×10^{-14}	5.74×10^{-08}	26.22	17.91
1000	3004	9	0	-4.274	1.38×10^{-13}	3.00×10^{-07}	39.37	29.70
1250	3754	9	0	-4.274	3.97×10^{-13}	2.25×10^{-08}	55.52	44.54
1500	4504	9	0	-4.274	2.02×10^{-13}	2.73×10^{-07}	74.21	62.14
1750	5254	9	0	-4.274	3.97×10^{-13}	7.22×10^{-08}	92.85	79.83
2000	6004	8	0	-4.273	5.57×10^{-13}	8.53×10^{-07}	103.15	88.00
2250	6754	9	0	-4.273	5.77×10^{-13}	6.85×10^{-08}	131.35	113.28
2500	7504	10	0	-4.273	4.58×10^{-13}	5.03×10^{-07}	194.39	171.67

solves the cam problem without any difficulty for n up to 2500 and more. The E04NQF solver does have some difficulty with this problem.

For this problem it is therefore expected of the QP solvers to outperform the dual solver with a considerable increase in speed. This speed increase is, however, not as high as one would expect it to be. Notice in the last column how much time is wasted while waiting for a complete flag to be found by the competing threads. A small amount of time is also lost as overheads, although this is close to negligible when comparing the desired parallel time with the true serial execution time. (See desired speedup in Figure 5.1.)

However, the most promising result is not the most apparent without further knowledge of the iteration histories. On closer inspection it is seen that the iteration history for the E04NQF and LSQP solver differs from that of the parallel solver. The parallel solvers converge on average six iterations before the LSQP solver, the next most promising solver.

Iteration history for cam design problem

SAOi algorithm

Outer	Inner	Function val	Max constr	...	ActHi	ActC		Message
0	0	-4.7123890E+00	7.5000E-01					
1	0	-0.4337959E+01	0.5078E+00	...	0	2140	5	E04NQF
2	0	-0.4562090E+01	0.1880E+00		0	2221	5	E04NQF
3	0	-0.4725032E+01	0.4574E-03		1	2000	5	E04NQF
4	0	-0.4645652E+01	0.1796E-03	...	4	2303	5	E04NQF
5	0	-0.4301003E+01	0.3085E-04		2	2476	5	E04NQF
6	0	-0.4270891E+01	0.1791E-07		37	2596	0	E04NQF
7	0	-0.4273425E+01	0.8535E-10		128	2596	0	LSQP
8	0	-0.4273427E+01	0.5573E-12	...	128	2596	0	LSQP

n = 2000

In Appendix F and in the extract above it is clear that neither solver is preferred for this problem, but that a combination of solvers is used. This is explained by the way in which the problem properties change during the search. The solver best suited for the subproblem is used at each step. For this problem, the `E04NQF` solver is preferred until a certain number of active constraints is satisfied, after which the `LSQP` routine takes over the final iterations.

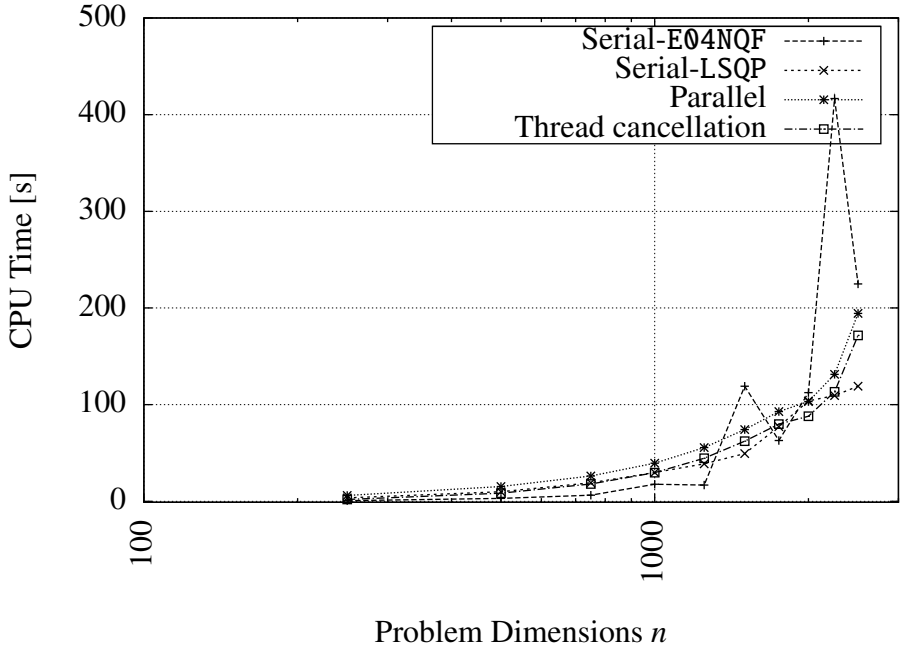


Figure 5.1: Maximization problem for the area of a valve opening for one rotation of a convex cam and solved using dual and QP solvers in both serial and parallel.

5.4.2 Svanberg's *n*-variate cantilever beam

Svanberg's *n*-variate cantilever beam is discussed in Section 4.4.4 and is used here without variation.

In contrast to the previous problem, one observes that the execution time is considerably less for the dual solver. This correlates with the findings of Etman *et al.* (2009). The increase in the solution time of the parallel version is attributed to the time spent waiting for the flags to indicate that the subproblems have completed. Once again the parallel solver exits with the least amount of outer iterations.

The iteration history below shows how the approximation properties stay constant as the as the solution progresses. The `l-BFGS-b` routine performs best for this type of problem, as the dual approximation takes advantage of the single constraint.

Table 5.5: Serial execution of Svanberg's n-variate cantilever beam problem–`l-BFGS-b`

n	m	k^*	l^*	f^*	h^*	r^*	CPU
250	1	7	0	1.310	-5.01×10^{-08}	2.19×10^{-08}	0.01
500	1	8	0	1.310	-2.00×10^{-10}	8.74×10^{-11}	0.01
750	1	8	0	1.310	-5.02×10^{-09}	2.19×10^{-09}	0.01
1000	1	8	0	1.310	8.40×10^{-10}	3.67×10^{-10}	0.02
1250	1	8	0	1.310	-4.34×10^{-10}	1.89×10^{-10}	0.02
1500	1	8	0	1.310	-7.11×10^{-09}	3.11×10^{-09}	0.02
1750	1	9	0	1.310	3.04×10^{-08}	1.33×10^{-08}	0.03
2000	1	9	0	1.310	-1.03×10^{-10}	4.52×10^{-11}	0.03
2250	1	9	0	1.310	-3.21×10^{-10}	1.40×10^{-10}	0.04
2500	1	9	0	1.310	-6.42×10^{-10}	2.80×10^{-10}	0.04

Table 5.6: Serial execution of Svanberg's n-variate cantilever beam problem–`E04NQF`

n	m	k^*	l^*	f^*	h^*	r^*	CPU
250	1	8	0	1.310	2.22×10^{-16}	1.12×10^{-05}	0.18
500	1	8	0	1.310	5.04×10^{-14}	2.00×10^{-05}	1.00
750	1	8	0	1.310	3.98×10^{-13}	1.91×10^{-05}	3.36
1000	1	8	0	1.310	1.37×10^{-12}	2.35×10^{-05}	8.65
1250	1	9	0	1.310	2.00×10^{-15}	2.07×10^{-05}	14.01
1500	1	9	0	1.310	3.55×10^{-15}	2.46×10^{-05}	23.81
1750	1	9	0	1.310	9.55×10^{-15}	3.11×10^{-05}	38.82
2000	1	9	0	1.310	6.68×10^{-14}	2.66×10^{-05}	55.45
2250	1	9	0	1.310	1.33×10^{-13}	3.41×10^{-05}	446.77
2500	1	9	0	1.310	3.49×10^{-13}	4.15×10^{-05}	541.94

Table 5.7: Serial execution of Svanberg's n-variate cantilever beam problem–`LSQP`

n	m	k^*	l^*	f^*	h^*	r^*	CPU
250	1	7	0	1.310	1.36×10^{-11}	1.79×10^{-11}	0.02
500	1	8	0	1.310	-1.02×10^{-12}	1.58×10^{-10}	0.05
750	1	8	0	1.310	-3.92×10^{-14}	5.39×10^{-10}	0.07
1000	1	8	0	1.310	7.86×10^{-13}	1.60×10^{-09}	0.10
1250	1	8	0	1.310	1.03×10^{-12}	4.42×10^{-11}	0.13
1500	1	8	0	1.310	-4.26×10^{-14}	9.93×10^{-11}	0.15
1750	1	8	0	1.310	8.42×10^{-12}	1.06×10^{-10}	0.17
2000	1	9	0	1.310	-5.28×10^{-14}	3.59×10^{-10}	0.23
2250	1	9	0	1.310	-7.59×10^{-14}	5.74×10^{-10}	0.27
2500	1	9	0	1.310	-9.07×10^{-14}	7.36×10^{-10}	0.28

Table 5.8: Parallel execution of Svanberg's n-variate cantilever beam problem.

n	m	k^*	l^*	f^*	h^*	r^*	CPU	CPU$_d$
250	1	6	0	1.310	1.46×10^{-07}	4.34×10^{-07}	0.31	0.01
500	1	7	0	1.310	3.57×10^{-10}	1.46×10^{-09}	2.87	0.04
750	1	7	0	1.310	1.11×10^{-08}	5.66×10^{-08}	5.46	0.04
1000	1	7	0	1.310	1.82×10^{-08}	1.08×10^{-07}	9.97	0.05
1250	1	7	0	1.310	5.98×10^{-09}	3.91×10^{-08}	16.38	0.07
1500	1	7	0	1.310	1.82×10^{-09}	8.96×10^{-09}	24.41	0.08
1750	1	7	0	1.310	3.07×10^{-08}	2.06×10^{-07}	37.12	0.08
2000	1	7	0	1.310	9.72×10^{-08}	6.76×10^{-07}	52.14	0.10
2250	1	8	0	1.310	-7.97×10^{-08}	3.48×10^{-08}	433.53	0.11
2500	1	8	0	1.310	4.77×10^{-10}	4.39×10^{-09}	530.59	0.12

Iteration history for n-variate cantilever problem

SAOi algorithm

Outer	Inner	Function val	Max constr	...	ActHi	ActC	Message
0	0	1.5600000E+00	2.2204E-16				
1	0	0.1316960E+01	0.1110E+00	...	0	1	0 BFGS
2	0	0.1306694E+01	0.2392E-01		0	1	0 BFGS
3	0	0.1309841E+01	0.2990E-02		0	1	0 BFGS
4	0	0.1310269E+01	0.3507E-03	...	0	1	0 BFGS
5	0	0.1310313E+01	0.4382E-04		0	1	0 BFGS
6	0	0.1310320E+01	0.6528E-05		0	1	0 BFGS
7	0	0.1310322E+01	0.9722E-07	...	0	1	0 BFGS

n = 2000

For this problem, the loss of performance is much higher than for the cam problem. In this case, most of the real-time recorded is the time spent waiting for the competing routines. This problem shows how language-level thread cancellation would benefit irregular algorithms such as these. Figure 5.2 clearly indicates the speed increase possible when solving the subproblems in parallel.

5.4.3 Vanderplaats' cantilever beam with tip displacement constraint

Vanderplaats (2001) proposed the optimal sizing design of the tip-loaded multi-segmented cantilever beam. The beam is of fixed length l, is divided into p segments, and is subject to geometric, and stress constraints and a single displacement constraint. The geometry has been chosen such that a very large number of the constraints are active or 'near-active' at the optimum. The complete problem statement is given in Section 4.4.3. The additional tip displacement constraint

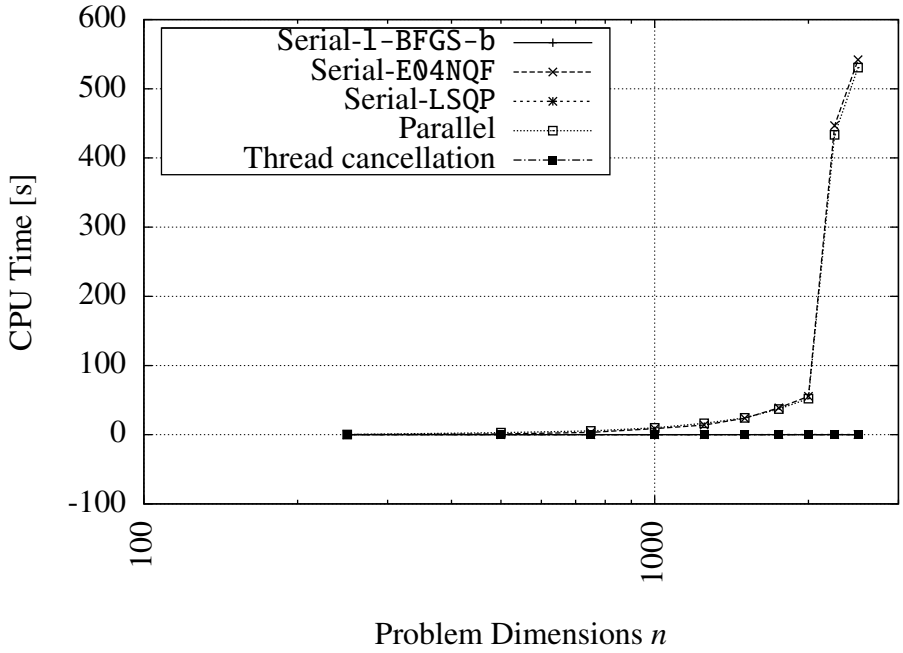


Figure 5.2: Serial and parallel execution of Svanberg's n-variate cantilever beam problem.

is

$$\frac{u_{tip}(\boldsymbol{b},\boldsymbol{h})}{\bar{u}} - 1 \leq 0.$$

Table 5.9: Serial execution of Vanderplaats' cantilever beam problem–`l-BFGS-b`.

n	m	k^*	l^*	f^*	h^*	r^*	CPU
500	501	12	0	63667.759	1.34×10^{-06}	4.26×10^{-02}	1.01
1000	1001	12	0	63665.582	1.33×10^{-06}	4.24×10^{-02}	2.56
1500	1501	12	0	63665.331	1.26×10^{-06}	4.01×10^{-02}	5.57
2000	2001	13	0	63665.215	1.28×10^{-06}	4.07×10^{-02}	9.96
2500	2501	14	0	63664.974	6.98×10^{-06}	2.22×10^{-01}	15.01
3000	3001	13	0	63665.133	1.25×10^{-06}	3.98×10^{-02}	21.96
3500	3501	13	0	63665.100	3.69×10^{-06}	5.23×10^{-02}	28.14
4000	4001	13	0	63665.142	3.07×10^{-09}	9.76×10^{-05}	38.39
4500	4501	13	0	63664.877	8.16×10^{-06}	2.60×10^{-01}	44.40
5000	5001	14	0	63664.907	6.99×10^{-06}	2.22×10^{-01}	66.41

This problem demonstrates how dual and QP solvers can be equally effective in solving approximations with similar properties (see Figure 5.3). There is little performance difference between the `LSQP` and the `l-BFGS-b` routines for this problem. The `E04NQF` routine, however, struggles greatly to solve this problem. One may therefore assert that the performance between different QP solvers varies just as greatly as between dual and QP solvers. The iteration history below

Table 5.10: Serial execution of Vanderplaats' cantilever beam problem–`E04NQF`.

n	m	k^*	l^*	f^*	h^*	r^*	CPU
500	501	13	0	63667.802	9.95×10^{-14}	4.67×10^{-05}	1.04
1000	1001	13	0	63665.624	1.28×10^{-13}	3.21×10^{-05}	6.34
1500	1501	13	0	63665.371	7.82×10^{-14}	2.55×10^{-05}	18.94
2000	2001	13	0	63665.256	6.39×10^{-14}	3.30×10^{-05}	31.83
2500	2501	13	0	63665.196	7.11×10^{-14}	6.22×10^{-05}	128.86
3000	3001	13	0	63665.173	6.39×10^{-14}	6.22×10^{-05}	224.95
3500	3501	12	0	63665.153	7.11×10^{-14}	7.66×10^{-05}	373.77
4000	4001	12	0	63665.142	7.11×10^{-14}	8.37×10^{-05}	545.20
4500	4501	13	0	63665.136	8.88×10^{-14}	1.08×10^{-04}	1904.90
5000	5001	13	0	63665.129	7.11×10^{-14}	1.39×10^{-04}	3139.57

Table 5.11: Serial execution of Vanderplaats' cantilever beam problem–`LSQP`.

n	m	k^*	l^*	f^*	h^*	r^*	CPU
500	501	13	0	63667.802	7.88×10^{-14}	3.01×10^{-05}	0.68
1000	1001	13	0	63665.624	1.91×10^{-13}	2.11×10^{-05}	2.04
1500	1501	13	0	63665.371	2.37×10^{-13}	1.72×10^{-05}	4.14
2000	2001	13	0	63665.256	2.67×10^{-13}	1.49×10^{-05}	6.99
2500	2501	13	0	63665.196	3.49×10^{-13}	1.33×10^{-05}	10.79
3000	3001	13	0	63665.173	4.39×10^{-13}	1.21×10^{-05}	15.63
3500	3501	13	0	63665.153	8.27×10^{-13}	1.12×10^{-05}	21.65
4000	4001	13	0	63665.142	6.00×10^{-14}	1.05×10^{-05}	29.62
4500	4501	13	0	63665.136	1.47×10^{-13}	9.90×10^{-06}	39.76
5000	5001	13	0	63665.129	1.46×10^{-13}	9.39×10^{-06}	54.38

Table 5.12: Parallel execution of Vanderplaats' cantilever beam problem.

n	m	k^*	l^*	f^*	h^*	r^*	CPU	CPU_d
500	501	12	0	63667.707	2.98×10^{-06}	9.48×10^{-02}	3.47	1.34
1000	1001	12	0	63665.540	2.63×10^{-06}	8.37×10^{-02}	17.48	4.23
1500	1501	12	0	63665.290	2.54×10^{-06}	8.06×10^{-02}	34.55	8.71
2000	2001	13	0	63665.176	2.51×10^{-06}	7.99×10^{-02}	61.74	15.82
2500	2501	13	0	63665.116	7.12×10^{-06}	7.99×10^{-02}	167.72	24.12
3000	3001	13	0	63665.094	2.50×10^{-06}	7.96×10^{-02}	284.49	32.06
3500	3501	13	0	63665.075	2.44×10^{-06}	7.76×10^{-02}	453.16	41.85
4000	4001	13	0	63665.064	2.46×10^{-06}	7.83×10^{-02}	637.75	52.99
4500	4501	13	0	63665.057	8.76×10^{-06}	7.87×10^{-02}	1069.92	67.87
5000	5001	13	0	63665.050	2.47×10^{-06}	7.86×10^{-02}	3889.09	86.41

shows how the approximation changes as the problem becomes feasible and how the `l-BFGS-b` routine takes over execution.

Iteration history for Vanderplaats' cantilever problem

SAOi algorithm

Outer	Inner	Function val	Max constr	...	ActC		Message
0	0	1.5000000E+05	-4.0476E-01				
1	0	0.5910025E+05	0.9355E+00	...	525	0	LSQP
2	0	0.5404018E+05	0.8151E+00		1582	0	LSQP
3	0	0.5908101E+05	0.1761E+00		1039	0	LSQP
4	0	0.6333578E+05	0.1608E-01	...	6	0	BFGS
5	0	0.6365954E+05	0.3002E-03		5	0	BFGS
6	0	0.6366509E+05	0.1573E-04		223	0	BFGS
7	0	0.6366525E+05	0.2143E-04	...	872	0	BFGS
8	0	0.6366516E+05	0.3053E-05		977	0	BFGS
9	0	0.6366518E+05	0.2429E-05		977	0	BFGS
10	0	0.6366518E+05	0.2531E-05		977	0	BFGS
11	0	0.6366518E+05	0.2511E-05	...	977	0	BFGS
12	0	0.6366518E+05	0.2515E-05		977	0	BFGS
13	0	0.6366518E+05	0.2514E-05		977	0	BFGS

n = 2000

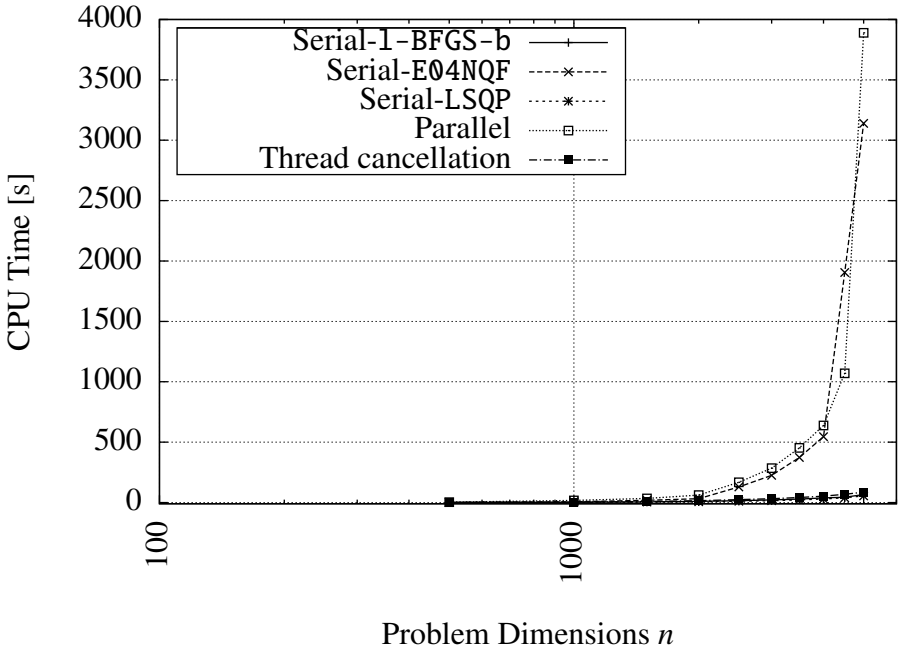


Figure 5.3: Serial and parallel execution of Vanderplaats' cantilever beam problem.

5.5 Summary

A parallel SAO method was used to show that it is beneficial to solve the subproblems with multiple subsolvers in parallel. Several problems with thread cancellation in OpenMP were outlined. Additionally, it was demonstrated how the use of flagged thread cancellation is detrimental to the performance of the routine.

Problems tend to suffer unnecessarily from long wait times as a consequence of the competing routines struggling to reach the checkpoints. The greatest deficiency was the difficulty of checking the flags at sufficiently short intervals in order to minimize the time that one algorithm waits for the other to return. The more "difficult" or larger a problem becomes for a certain routine, the more time is spent waiting for that flag to be set.

Another point that is noteworthy is that all outer iterations do not necessarily prefer one single solver, which results in a mixture of routines in the program history. The program run histories are shown in the individual sections. The QP subroutines are used interchangeably for the cam problem. This shows that the performance of the solvers is dependent on certain problem properties, although solvers of the same type can perform equally well on any given iteration if the solvers have similar performance for that problem type. The dual solver is preferred for the cantilever problem. In the Vanderplaats' cantilever beam problem one can see that the dual `l-BFGS-b` and QP `LSQP` solvers are used interchangeably. Thus, it is clear that there are certain approximations for which dual and QP solvers perform equally well.

The number of iterations needed by the parallel algorithm to complete is less than that for the serial algorithms in all cases. This has profound implications for FEM and/or CFD programs, as many function evaluations are done between outer iterations. Therefore, by reducing the number of outer iterations, the performance of these FEM and CFD models will ultimately be increased substantially.

In conclusion, the proposed parallel dual versus QP method is a very promising alternative when the problem type is not known in advance. This approach automatically chooses the best solver on each approximation, which benefits the solution of all structural optimization problems. As expected, the dual method remains the most efficient for problems with only a few constraints, but the QP method is still superior for solving problems with many design variables and constraints.

Chapter 6

Parallel global algorithms

In structural optimization, function evaluations typically involve a complete finite element (FE) or boundary element (BE) analysis. By spreading the independent local minimization steps across multiple CPUs it is possible to determine the number of CPUs that will optimally reduce the cost of a multi-start global optimization implementation in a shared memory paradigm.

6.1 Background

In engineering design, the optimum design is not always realizable. This could be a consequence of physical and/or cost constraints. Therefore, knowledge of complementary solutions is especially advantageous. However, classical optimization techniques, such as sequential approximate optimization (SAO) and sequential quadratic programming (SQP), require multiple restart points and multiple runs in order to find similar solutions, with no guarantee that these will yield a practicable design solution. Global optimization deals with the optimization task involved in finding all or most of the multiple solutions in multimodal problems.

The SAO*i* algorithm is generally aimed at unimodal problems, but also incorporates the option to optimize smooth multimodal problems. This is achieved by a multi-start strategy in combination with the Bayesian acceptance condition[1] realised by Snyman and Fatti (1987).

In Bolton *et al.* (2000) it is shown that multiple independent searches in a multi-start procedure, performed simultaneously on different computers, effectively reduce the cost of solving expensive global optimization problems using a PVM (Parallel Virtual Machine) with 128 processors (PVM is similar to the more modern MPI implementations). A multi-start strategy coupled with the Bayesian stopping condition is implemented using a simple parallel OpenMP algorithm. By varying the number of CPUs from one to eight the number of CPUs that will optimally reduce the cost of this implementation is determined. The attraction of OpenMP for this type of implementation is the ease with which the algorithm may be parallelized and that it may be run equally well on one or more CPUs without modification of the algorithm.

[1]The proof of the Bayesian stopping condition can be shown to be a generalization of the procedure proposed by Zielinsky (1981); the proof is included in Section C and is also found in Snyman *et al.* (2003)

Section 6.2 defines multimodal optimization when applied in conjunction with the Bayesian stopping condition. The multithreaded parallel implementation is described in Section 6.3 and is applied to two global optimization problems, described in Section 6.4.

6.2 Multimodal optimization with the Bayesian stopping condition

The simplest global optimization algorithm is most likely one developed by combining multiple local searches with some probabilistic stopping criterion. Multi-start methods require such a termination rule for deciding when to end sampling. The overall minimum function value $\tilde{f}$ is then chosen as the approximation to the global minimum f^*, i.e.

$$\tilde{f} = \min \left\{ \hat{f}^j,\ j = 1, 2, \cdots \right\}, \tag{6.2.1}$$

where j represents the number of starting points to date, and all $\hat{f}_j$ are assumed to be feasible local minima for $j = 1, 2, \cdots$. Snyman *et al.* (2004) demonstrate the suitability of the Bayesian stopping criterion for solving two sets of global unconstrained minimization problems. The initial set is a set of small-sized and relatively simple test functions, followed by a set of larger and more challenging problems.

In the Bayesian acceptance condition, the only assumption made is that the region of the attraction of the global optimum is comparable to, or larger than, the region of attraction of any other local optimum (Bolton *et al.*, 2000). The approach is simple: the automated multi-start strategy is terminated when the probability of convergence to the global optimum is larger than, or equal to, some desired confidence level q^*.

Now the region of convergence of a local minimum $\hat{\boldsymbol{x}}$ is defined as the set of all points $\boldsymbol{x}$ which, when used as starting points for a given algorithm, result in converge to $\hat{\boldsymbol{x}}$. Let R^k denote the region of convergence of local minimum $\hat{\boldsymbol{x}}^k$ and let α^k be the associated probability that a sample point be selected in R^k. The region of convergence and the associated probability for the global minimum $\boldsymbol{x}^*$ are denoted by R^* and α^* respectively. The following basic assumption is now made (Bolton *et al.*, 2000):

$$\alpha^* \geq \alpha^k \text{ for all local minima } \hat{\boldsymbol{x}}^k\ . \tag{6.2.2}$$

Theorem 1 *Furthermore, let r be the number of sampling points falling within the region of convergence of the overall best minimum $\tilde{f}$ after $\tilde{n}$ points have been sampled. Then, under assumption* (6.2.2)*, the probability that $\tilde{f}$ corresponds to f^* is given by*

$$Pr\left[\tilde{f} = f^*\right] \geq q(\tilde{n}, r) = 1 - \frac{(\tilde{n}+1)!\ (2\tilde{n}-r)!}{(2\tilde{n}+1)!\ (\tilde{n}-r)!}, \tag{6.2.3}$$

where Pr is short for 'probability that'.

Based on Theorem 1, the algorithm will stop when

$$\Pr\left[\tilde{f} = f^*\right] \geq q^*. \tag{6.2.4}$$

6.3 Parallel implementation and algorithm specifics

When the CPU requirements of evaluating f are significantly higher than forking problem-specific data and algorithm internals to multiple CPUs, the overheads required for a specific job become inappreciable. Few analytic example problems exist that mirror the high cost involved in evaluating f. By simulating the high cost of an FE call using Algorithm 4 it is possible to determine the effectiveness of our parallel implementation for problems that resemble real-world examples.

Algorithm 4: Simulation of costly FE calls.

```
  !Function cost increases when foo is increased.
1 foo = 100
2 bar = 0
3 do i = 1,foo
4     do j = 1,foo
5         bar = bar + 1
6     end do
7 end do
```

Although analytical unconstrained problems are more common and thoroughly tested (Arora, 1990), many practical global optimization problems include explicit constraints. Compared to the unconstrained case, there are relatively few algorithms for solving the constrained global optimization problem (Snyman *et al.*, 2003). Furthermore, Snyman *et al.* (2003) argue the unified Bayesian stopping rule to be as applicable to the constrained algorithms as the unconstrained algorithms. The constrained global programming problem is addressed by applying the above, essentially local, optimizers in the multi-start strategy described by Snyman and Fatti (Snyman and Fatti, 1987). Either dual or QP solvers may be used for solving such problems. The `LSQP` routine proved the most promising and is therefore selected for demonstration purposes.

The number of function evaluations associated with each search determine the cost of the global search. Given that the sampling steps are randomly generated the number of function evaluations may vary greatly between respective repetitions, depending on the starting points. Snyman *et al.* (2003) and Bolton *et al.* (2000) define the apparent visual cost N_{vc} as the number of function evaluations associated with the 'longest' search on a given CPU. This is not a suitable assumption when one evaluates the effectiveness of the parallel algorithm, because the search trajectory for each run is different. For comparison, the average cost N_{avg} for both N_{fe} and N_{ge} is calculated to the global optimum. This is done for 100 random restarts of each algorithm for the complete test set. The execution time associated with the search trajectories is also averaged,

and is denoted as CPU_{avg}. This includes the time ascribed to the initialization and evaluation of the stopping criterion (6.2.4).

In contrast to the implementation used by Bolton *et al.* (2000), a simple shared memory model is proposed that can easily scale to run on one or more processors. This implementation is realised using the fork-join shared memory model in OpenMP.

6.3.1 OpenMP implementation

A parallel shared memory algorithm is implemented within the `split.f` file of the SAO*i* program. This turns the call to the local minimizers within the secondary SAO*i* driver into a parallel call. The OpenMP directives are chosen with care to ensure the thread safety of the algorithm.

In the previous chapter it was learned that it is impossible to branch out of OpenMP loops, with the !$omp **do** loop construct limited to loops in which the number of iterations can be counted (Chapman *et al.*, 2008). The need for thread cancellation in OpenMP is once again emphasized. Fortunately, in this case, the workaround is more effective. After convergence, it is possible to encapsulate all further computations within the !$omp **parallel** construct an `if` statement. This prevents any further computational cost to be lost to unnecessary computations, and all iterations therefore complete before exiting the loop construct safely. Algorithm 5 describes the parallel OpenMP environment.

The Bayesian stopping condition requires some level of sequential operation in order to correctly calculate the probability of convergence; the sequence in which iterations complete must be known without simultaneous writing of data to variables. These variables are the number of successful random searches $\tilde{n}$, and the number of starting points r (from which convergence to the current best minimum $\tilde{f}$ occurs) is used to calculate the probability $q(\tilde{n}, r)$. By using structured blocks, which are OpenMP-defined loop constructs, structured updating of shared variables can be enforced. Two of these loop constructs are applicable here (Chapman *et al.*, 2008):

1. !$omp **ordered**, which ensures that the corresponding block of code is executed in the order of the loop iterations, and

2. !$omp **critical**, which ensures that no two threads execute the piece of code in question simultaneously, on a first come, first served basis.

The critical region is expensive to implement because the runtime system needs to keep track of the iterations that have completed, and possibly keep certain threads in a wait state until their results are needed. Nevertheless, the critical region is well suited for code where the actual order in which threads perform the computation is not important. One must keep in mind that the size of the critical region affects the potential wait times. Program performance might be poor for particularly large critical regions. Compared to the computational expense of performing a local minimization step, the cost of assessing global convergence is negligible.

The parallel global algorithm is presented in Algorithm 5.

Algorithm 5: Parallel global optimization algorithm for SAO*i*.

```
   !unset flag
 1 flag = 0
 2 omp_set_thread_num(proc)
 3 !$ omp parallel default(fistprivate)
 4     !$ omp+shared (it,fopt,xopt,xkktopt,vopt,popt,nfe,nge,nthreads,···
 5     !$ omp do schedule(dynamic)
 6         do it = 1,itglobalmax
 7             if (flag.ne.1) then
 8                 it = omp_get_thread_num()
                   !construct a random starting point (a vectorized call)
 9                 do i=1,n
10                     call ranmar(rdm,1,u,c,cd,cm,i97,j97,raninit)
                       x(i) = (x_u(i) − x_l(isec : global)) ∗ (rdm) + x_l(i)
11                 end do
                   !secondary driver
12                 call SAOi_split(n,ni,ne,x,···)
13                 !$ omp critical
                       ⋮
14
15                     if (f.lt.fopt.and.feasible) then
16                         fopt=f
17                         ir = ir+1
18                         xopt=x
19                         xkktopt = xkkt
20                         p = 1.0d0-conv(it,ir)
21                     end
                       !successful search?
22                     if (p.gt.ptarget) then
23                         flag = 1
24                     end
25                 !$ omp end critical
26             end
27         end do
28     !$ omp end do nowait
29 !$ omp end parallel
```

6.4 Results

The first problem to be evaluated is an unconstrained global optimization problem, and the second is a constrained global optimization problem. Both the QP and dual solvers are available, but the LSQP solver is the most effective and is therefore used here.

The computational effort is averaged over 100 independent search trajectories, each associated with a random starting point $\boldsymbol{x}_0^j$. This is then repeated for an increasing number of CPUs, $c = [0, 1, \cdots, 8]$.

6.4.1 Griewank

$$\min_{\boldsymbol{x}} f(\boldsymbol{x}) = 1 + \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) \tag{6.4.1}$$

subject to $-600 \leq x_i \geq 600, i \in \{1, 2, \cdots, n\}$. The function has a global minimum located at $x^* = \{0, 0, \cdots, 0\}$ with $f(x^*) = 0$. The number of local minima for arbitrary n are unknown, but in the two-dimensional case there are some 500 local minima. Tests are performed for $n = 2, 10$.

During execution, some of the calculated trajectories for the Griewank problem result in the local minima at 1.45×10^{-01}. This can be avoided by increasing the probability of convergence. This occasionally occurs for Pr= 0.990. Take heed that the resultant function values of 10^{-08} and smaller are sufficiently close to zero (the global minimum) according to the machine precision.

Table 6.1: Computational effort for Griewank problem, two dimensions.

c	n	k^*	f^*	r^*	$N_f e$	$N_g e$	Pr	CPU_{avg}
1	2	0	7.6347251×10^{-10}	3.9×10^{-05}	4718	2810	0.992 > 0.990	0.85
2	2	0	1.4561869×10^{-01}	1.3×10^{-05}	4597	2735	0.993 > 0.990	0.53
3	2	0	1.4561869×10^{-01}	1.3×10^{-05}	4691	2800	0.993 > 0.990	0.44
4	2	0	2.0927704×10^{-13}	4.6×10^{-07}	4888	2916	0.992 > 0.990	0.43
5	2	0	1.3544721×10^{-14}	1.1×10^{-07}	4598	2748	0.992 > 0.990	0.43
6	2	0	2.6645353×10^{-15}	5.2×10^{-08}	4908	2925	0.992 > 0.990	0.48
7	2	0	1.1102230×10^{-16}	9.5×10^{-09}	4668	2781	0.992 > 0.990	0.47
8	2	0	9.6846198×10^{-11}	9.9×10^{-06}	4411	2627	0.996 > 0.990	0.47

6.4.2 Part-stamp

The objective of this stamping problem is to minimize the area of a rectangular plate needed to stamp out a collection of n_d disks of given sizes. These disks may not overlap (Mulkay and Rao, 1998), as shown in Figure 6.1. The number of variables and the number of constraints are

Table 6.2: Computational effort for Griewank problem, 10 dimensions.

c	n	k^*	f^*	r^*	N_fe	N_ge	Pr	CPU_{avg}
1	10	0	1.5106833×10^{-08}	5.8×10^{-05}	1396	370	0.993 > 0.990	0.27
2	10	0	2.1256841×10^{-10}	1.1×10^{-05}	1545	414	0.995 > 0.990	0.19
3	10	0	8.9873232×10^{-09}	4.4×10^{-05}	1592	426	0.999 > 0.990	0.14
4	10	0	2.1684203×10^{-09}	2.2×10^{-05}	1572	418	0.996 > 0.990	0.12
5	10	0	2.4561871×10^{-09}	2.2×10^{-05}	1891	506	0.991 > 0.990	0.14
6	10	0	1.3322676×10^{-12}	8.3×10^{-07}	1902	507	1.000 > 0.990	0.14
7	10	0	7.6710849×10^{-10}	1.5×10^{-05}	2022	541	1.000 > 0.990	0.16
8	10	0	1.3942847×10^{-11}	3.6×10^{-06}	2075	553	1.000 > 0.990	0.17

Table 6.3: Computational effort for Griewank problem with added expense during the function evaluations, two dimensions.

c	n	k^*	f^*	r^*	N_fe	N_ge	Pr	CPU_{avg}
1	2	0	7.6347251×10^{-10}	3.9×10^{-05}	4718	2810	0.992 > 0.990	9.50
2	2	0	3.4416914×10^{-15}	5.9×10^{-08}	4405	2632	0.993 > 0.990	4.51
3	2	0	7.0114570×10^{-10}	3.7×10^{-05}	4942	2950	0.993 > 0.990	3.41
4	2	0	2.9904967×10^{-12}	2.4×10^{-06}	4724	2822	0.993 > 0.990	2.50
5	2	0	6.9949890×10^{-10}	3.7×10^{-05}	4905	2915	0.992 > 0.990	2.56
6	2	0	1.4561869×10^{-01}	5.9×10^{-06}	4700	2808	0.997 > 0.990	2.45
7	2	0	1.9984014×10^{-15}	4.6×10^{-08}	5014	2991	0.992 > 0.990	2.61
8	2	0	1.1102230×10^{-16}	9.2×10^{-09}	4721	2813	0.993 > 0.990	2.55

Table 6.4: Computational effort for Griewank problem with added expense during the function evaluations, 10 dimensions.

c	n	k^*	f^*	r^*	N_fe	N_ge	Pr	CPU_{avg}
1	10	0	1.5106833×10^{-08}	5.8×10^{-05}	1396	370	0.993 > 0.990	2.84
2	10	0	1.9818974×10^{-09}	3.0×10^{-05}	1476	392	0.994 > 0.990	1.58
3	10	0	1.2453042×10^{-09}	4.4×10^{-05}	1617	430	0.997 > 0.990	1.19
4	10	0	5.9133888×10^{-09}	3.4×10^{-05}	1713	455	0.995 > 0.990	1.00
5	10	0	2.6153862×10^{-09}	2.6×10^{-05}	1799	478	0.999 > 0.990	1.07
6	10	0	4.0338433×10^{-10}	1.6×10^{-05}	1933	519	0.999 > 0.990	1.15
7	10	0	1.2023238×10^{-10}	1.1×10^{-05}	2058	550	1.000 > 0.990	1.25
8	10	0	1.0658670×10^{-09}	1.4×10^{-05}	2047	547	1.000 > 0.990	1.27

easily scaled and therefore make this problem the ideal scalable constrained global optimization problem. The problem can be formulated as follows:

$$\begin{aligned} \min \quad & ab \\ \text{subject to} \quad & \begin{cases} x_i + R_i - a \leq 0, & i = 1, \cdots, n_d, \\ y_i + R_i - b \leq 0, & i = 1, \cdots, n_d, \\ R_i - x_i \leq 0, & i = 1, \cdots, n_d, \\ R_i - y_i \leq 0, & i = 1, \cdots, n_d, \\ (R_i + R_j)^2 - (x_i - x_j)^2 - (y_i - y_j)^2 \leq 0, & i = 1, \cdots, n_d - 1, \\ & j = i + 1, \cdots, n_d. \end{cases} \end{aligned} \tag{6.4.2}$$

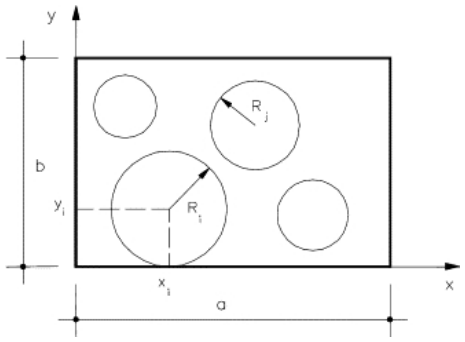


Figure 6.1: The stamping problem (Mulkay and Rao, 1998).

Table 6.5: Computational effort for part-stamp problem, 10 segments.

c	n	m	k^*	f^*	h^*	r^*	$N_f e$	$N_g e$	Pr	CPU_{avg}
1	22	65	0	40.000	-2.9×10^{-13}	5.7×10^{-08}	689	688	0.994 > 0.990	8.78
2	22	65	0	40.000	2.4×10^{-12}	4.1×10^{-08}	734	733	0.993 > 0.990	4.86
3	22	65	0	40.000	-4.1×10^{-13}	6.9×10^{-05}	683	682	0.994 > 0.990	3.08
4	22	65	0	40.000	-1.6×10^{-13}	2.3×10^{-06}	720	719	0.997 > 0.990	2.53
5	22	65	0	40.000	-1.4×10^{-13}	1.2×10^{-06}	743	741	0.998 > 0.990	2.53
6	22	65	0	40.000	-1.6×10^{-13}	6.8×10^{-06}	747	745	0.994 > 0.990	2.45
7	22	65	0	40.000	-3.0×10^{-13}	4.2×10^{-09}	820	819	0.993 > 0.990	2.59
8	22	65	0	40.000	-3.8×10^{-13}	8.8×10^{-08}	744	743	0.997 > 0.990	2.30

The disadvantages of the inability to cancel threads within an OpenMP loop construct is not as apparent here. Even though the residual threads within the loop are not needed, their execution has little influence on the performance.

Snyman *et al.* (2003) and Bolton *et al.* (2000) use a message-passing parallel model in which the workload is statically assigned in a master-slave configuration. The master assigns the tasks and interprets the results, while the slaves compute the search trajectories. Our approach is similar, although the implementation is multithreaded. This multithreaded approach is attractive

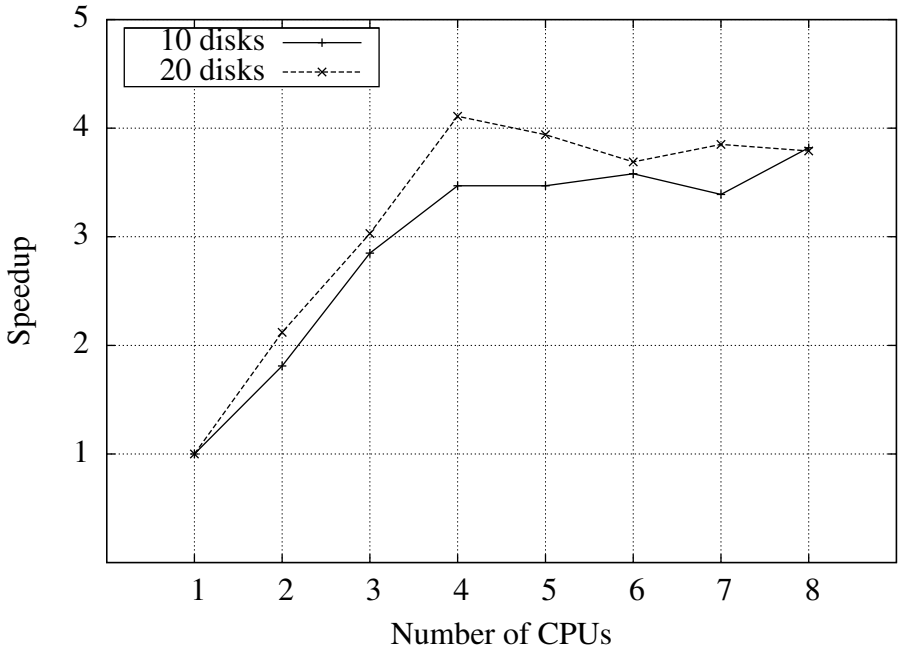


Figure 6.2: Speedups for the part-stamp global optimization problem for 10, 15 and 20 disks.

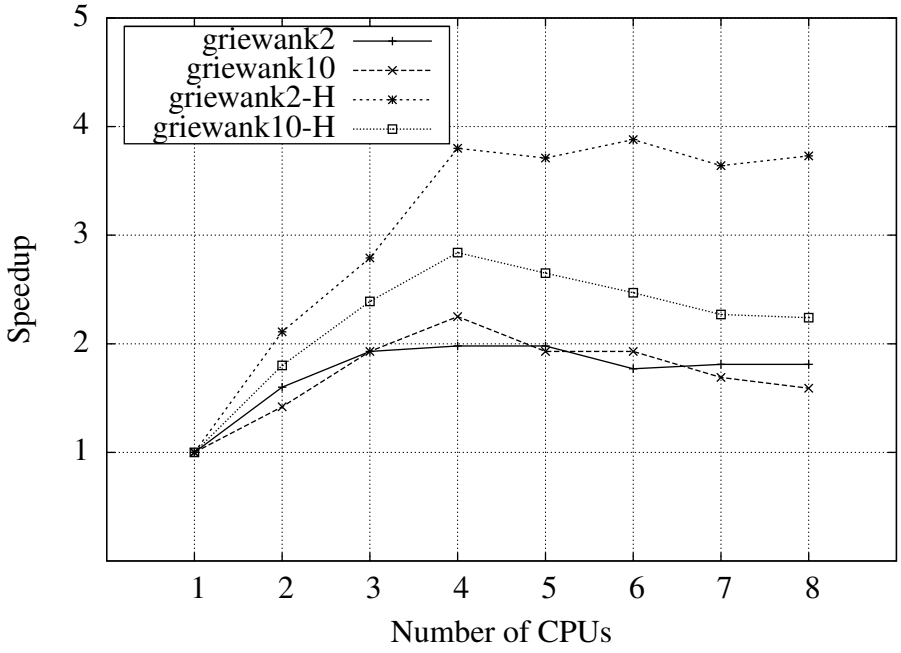


Figure 6.3: Speedups for global optimization problems.

Table 6.6: Computational effort for part-stamp problem, 20 segments.

c	n	m	k^*	f^*	h^*	r^*	$N_f e$	$N_g e$	Pr	CPU_{avg}
1	42	230	0	76.497	-3.1×10^{-13}	7.7×10^{-06}	1522	1482	0.993 > 0.990	306.53
2	42	230	0	76.497	4.0×10^{-12}	3.0×10^{-08}	1557	1513	0.993 > 0.990	144.26
3	42	230	0	76.497	-3.1×10^{-13}	3.2×10^{-05}	1612	1565	0.994 > 0.990	101.17
4	42	230	0	76.497	-2.6×10^{-13}	1.0×10^{-04}	1506	1461	0.993 > 0.990	74.66
5	42	230	0	76.497	6.0×10^{-13}	2.7×10^{-06}	1606	1562	0.997 > 0.990	77.88
6	42	230	0	76.497	3.0×10^{-13}	1.1×10^{-04}	1750	1700	0.997 > 0.990	83.17
7	42	230	0	76.497	7.1×10^{-13}	7.9×10^{-05}	1644	1596	0.997 > 0.990	79.60
8	42	230	0	76.497	-1.9×10^{-13}	1.1×10^{-04}	1675	1632	0.993 > 0.990	80.89

because of its simplicity and scalability. This implementation will work as well on multiple shared-memory CPUs as it does on one.

By not assigning an initial static number of threads, the algorithm terminates as soon as the desired probability of conversion is reached. This eliminates unnecessary continuation during the evaluation of the problem trajectory. Depending on the random starting points, the trajectory may only require a few iterations to complete. In this case, the statically assigned threads will greatly overestimate the probability of convergence and result in wasted work and processors. Conversely, if the trajectory requires many iterations, statically assigned threads may not find the global optimum to a sufficient probability.

Speed increases of up to four times are achieved by this implementation, making this approach reasonably attractive. With the problems showing increased efficiency as the problem complexity increases, these speed increases could become even greater. The performance levels out for more than four CPUs, a consequence of the layout of our eight cores in two clusters of four. Memory transfer operations are therefore more expensive.

6.5 Summary

The application of the probabilistic Bayesian stopping criterion proves to be applicable to multi-start constrained and unconstrained algorithms in parallel. The two standard test functions correctly reduce to their respective minimizers for all our available solvers. However, the `LSQP` solver proves to be the most effective for both the constrained and unconstrained cases.

Multithreaded parallelization proves to be a simple and effective method to reduce the computational time associated with the solution of both these global programming problems. While reducing the effective computational effort, the desired probability of convergence is reached. This approach makes effective use of the available computation units, although increasing the number of CPUs to more than four did not provide any additional performance in our case.

Chapter 7

Matrix-vector multiplication on the GPU

In this chapter, the acceleration of approximation-based optimization methods is studied by making use of accelerated dense matrix-vector products. These accelerated routines are based on the BLAS `dgemv` implementation, but are targeted to a specific hardware architecture, such as *multi-core* CPUs or *many-core* GPUs. Significant speedups are measured when evaluating the `dgemv` routine on the GPU – especially for large problem sizes.

7.1 Background

Approximation-based optimization methods rely on Taylor series expansions, which may be linear, quadratic, or even arbitrarily non-linear. The expansion used has profound implications on the computational effort required. Consider both the quadratic approximation $\tilde{f}(\boldsymbol{x})$ to the function $f(\boldsymbol{x})$ about the point $\boldsymbol{x}^{\{k\}}$, as well as the dual approximate subproblem $\beta_i(\boldsymbol{\lambda})$ as a function of $\tilde{f}(\boldsymbol{x})$.

The quadratic approximation $\tilde{f}(\boldsymbol{x})$ is written as

$$\tilde{f}(\boldsymbol{x}) = f^{\{k\}} + \boldsymbol{\nabla}^T f^{\{k\}} \boldsymbol{s} + \frac{1}{2} \boldsymbol{s}^T \boldsymbol{H}^{\{k\}} \boldsymbol{s}, \tag{7.1.1}$$

with $\boldsymbol{s} = (\boldsymbol{x} - \boldsymbol{x}^{\{k\}})$, and $\boldsymbol{\nabla}^T f^{\{k\}}$ and $\boldsymbol{H}^{\{k\}}$ the gradient vector and Hessian matrix of f at $\boldsymbol{x}^{\{k\}}$ respectively. For the sake of brevity, the abbreviated notation, $f^{\{k\}} = f(\boldsymbol{x}^{\{k\}})$, is used. Furthermore, $\boldsymbol{\nabla}^T f^{\{k\}}$ and $\boldsymbol{H}^{\{k\}}$ are constant and $\boldsymbol{H}^{\{k\}}$ is the fully populated Hessian matrix.

Similarly, the dual approximate subproblem from Section 2.2.3, where the inverse matrix-vector multiplication is required, is written as follows

$$\beta_i(\boldsymbol{\lambda}) = x_i^{\{k\}} - \left(c_{2i_0}^{\{k\}} + \sum_{j=1}^{m} \lambda_j c_{2i_j}^{\{k\}} \right)^{-1} \left(\frac{\partial f_0^{\{k\}}}{\partial x_i} + \sum_{j=1}^{m} \lambda_j \frac{\partial f_j^{\{k\}}}{\partial x_i} \right). \tag{7.1.2}$$

Equation (7.1.1) can be written as two matrix-vector products, with each of the BLAS-`gemv` routines forming $\boldsymbol{y} = \alpha \boldsymbol{A}\boldsymbol{x} + \beta \boldsymbol{y}$ equations, whereas (7.1.2) can be written as two transposed dense matrix-vector products, $\boldsymbol{y} = \alpha \boldsymbol{A}'\boldsymbol{x} + \beta \boldsymbol{y}$. Let us now assume that (7.1.1) and (7.1.2) are

to be evaluated during *some iterative process* for very many different vectors $\boldsymbol{x}$, such as in the approximation-based optimization method (SAO) considered here.

Profile data show that the calculation of the dense matrix-vector products using the BLAS-`dgemv` routines dominates the computational effort. Due to the significant contribution of the matrix-vector products to the total time required to find a solution, the performance of the `dgemv` routines supplied by several optimized BLAS libraries is considered, with the NetLib BLAS (Lawson *et al.*, 1979) used as the reference implementation. ATLAS BLAS (Whaley and Dongarra, 1998) and ACML (Advanced Micro Devices, Inc., 2010*a*) provide accelerated CPU-based implementations, whereas CUDA BLAS (CUBLAS) from NVIDIA (NVIDIA Corporation, 2010) is used to investigate GPU-based performance.

A significant amount of research has gone into solving very large, dense matrix-vector products. BLAS (Basic Linear Algebra Subprograms) (Dongarra *et al.*, 1988; Lawson *et al.*, 1979) has emerged as the standard interface with linear algebra libraries. ATLAS (Automatically Tuned Linear Algebra Software) BLAS (Whaley and Dongarra, 1998) is freely available, as are platform-optimized libraries, such as ACML (Advanced Micro Devices, Inc., 2010*a*) developed by AMD. Other libraries offered by AMD include ACML_mp which delivers multi-threaded support with the use of OpenMP. The fundamental routine is `dgemv` (double precision general matrix-vector multiplication). `Dgemv` is essential to approximation-based optimization methods that rely on Taylor series expansions.

Although GPUs typically have a much higher theoretical peak performance than CPUs (NVIDIA Corporation, 2008) (especially in the single precision case), measured performance improvements are often lower than expected when the overhead of transferring data to and from the device (which of course does not have unlimited memory) is taken into account.

In Section 2.2, the approximate subproblems (7.1.1) and (7.1.2) were introduced. It is aimed to optimize these routines by performing them on the GPU. Then the sizing design problem of the tip-loaded, multi-segmented cantilever beam proposed by Vanderplaats (2001) is introduced. A short introduction to GPU computing is provided in Section 7.2. Section 7.3.1 presents a number of benchmark results for the different BLAS implementations. In addition to the relative performance benchmarks of the different BLAS implementations considered, the influence on the performance of solving an optimal sizing design problem is investigated in Section 7.3.2. This work also investigates methods to reduce the impact of data transfer when these routines are used in the framework of numerical optimization. These methods include the intelligent reuse of data to minimize the required transfers. When such steps are taken, the GPU implementations are easily able to outperform CPU implementations, especially for very large problems.

7.2 GPU-based parallel programming

Parallel programming on graphic processing devices involves running a sequential *host* program on the *multi-core* CPU, as well as parallel *kernel* programs, which run on the CUDA or OpenCL-enabled graphics adapter (*many-core* GPUs) (Kirk and Hwu, 2010; NVIDIA Corporation, 2008). These highly parallel graphics processing devices run SPMD (Single Program Multiple Data) kernel programs, of which each computation is potentially executed on a very large number of

parallel threads.

High-end graphics-processing devices – such as the Nvidia GTX 280 – are built on arrays of SM (shared memory) multiprocessors, each of which is capable or supporting 1024 threads (Lindholm *et al.*, 2008). Every multiprocessor is equipped with eight scalar cores, 16384 32-bit registers and 16KB of high-bandwidth low-latency memory. The eight scalar cores perform integer and single precision floating point operations, while double precision floating point operations are handled by a single shared unit. It is ideal to take advantage of the device's ability to operate on large data sets (such as matrices), where the same operation can be performed across thousands of elements at the same time.

7.3 Results

This section presents a number of performance results. The first are for the matrix-vector products associated with (7.1.1) and (7.1.2) considered independently, followed by results for the design problem in question.

Performance results for `dgemv` BLAS routines are collected. These results are benchmarked in Section 7.3.1. In Section 7.3.2 these routines are applied to a real-world optimization problem discussed in Section 4.4.3; the additional displacement constraint is given in Section 5.4.3. The optimal sizing design of a cantilever beam proposed by Vanderplaats (2001) is solved with the use of four different optimized BLAS routines: NetLib BLAS, ATLAS BLAS, ACML and ACML_mp. Each has been chosen because of its inherent differences. NetLib BLAS is the system-independent implementation, and ATLAS BLAS is the platform-optimized BLAS version to take optimal advantage of the CPU cache. ACML is AMD's platform-optimized BLAS library, and ACML_mp is the multi-threaded version of the library. All implementations of these libraries are freely available.

7.3.1 Matrix-vector benchmarks

The results shown in Figure 7.1 are for two successive matrix-vector products in order to evaluate one complete quadratic approximation of (7.1.1) and (7.1.2) respectively. Note that the cost of data transfer to the device is not accounted for during the calls to the matrix-vector multiplications on the device. Figure 7.1 shows GFLOPS for each matrix-vector pair, increasing in problem size.

From Figure 7.1 it is clear that the matrix-vector routine for the GPU is superior if memory transfer is not taken into account. When the data transfer is taken into account, memory movement latencies have a detrimental influence on the performance, therefore these transfers of memory need to be minimized.

The bandwidth between the device memory and the compute units is much higher (141 GBps on the NVIDIA GeForce GTX 280) than the bandwidth between the host memory and the device memory (8 GBps on the PCI Express ×16 Gen2). Since all data to be acted on need to be transferred from the host to the device, these transfers need to be minimized (NVIDIA Corporation, 2009). Code that transfers data for brief use by a small number of threads is unlikely to see any

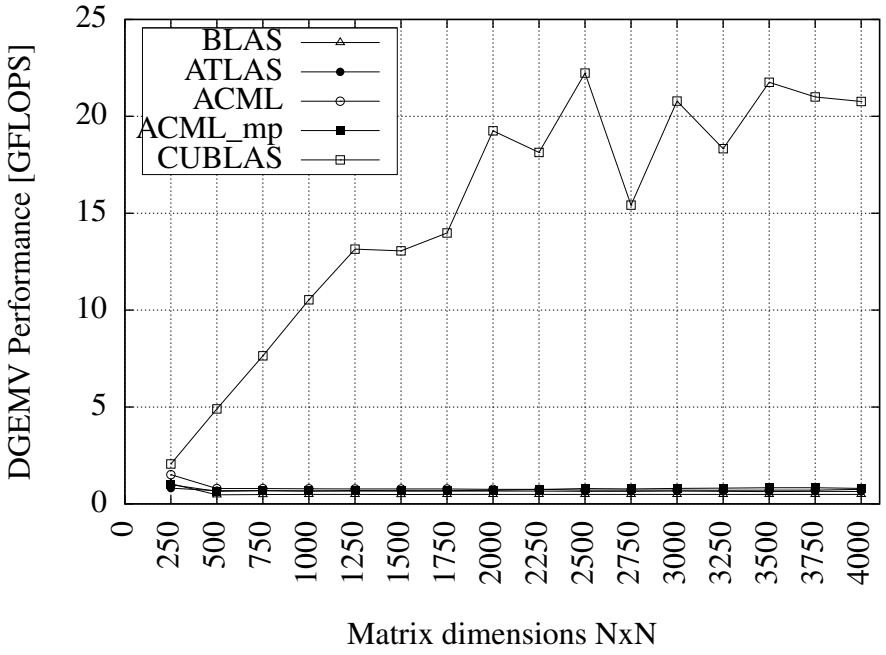


Figure 7.1: The performance in GFLOPS for two successive matrix-vector calls. Please note that the data transfers for the GPU version are excluded and that the ACML_mp data were collected for two cores.

performance lift. Hence, the complexity of the operations should justify the cost of moving the data to the device, because of the high computational throughput of the device. Accordingly, data should be kept on the device as long as possible. Running multiple kernels on the same data will favor leaving the data on the device between kernel calls. Figure 7.2 shows the cost involved in transferring $\boldsymbol{A}$, $\boldsymbol{x}$ and $\boldsymbol{y}$ compared to the transfer of only $\boldsymbol{x}$ and $\boldsymbol{y}$ to and from the device.

It is clear that copying the square matrix $\boldsymbol{A}$ is the most expensive operation. Moving $\boldsymbol{A}$ to device memory has an operational cost of $O(N^2)$. Hence, keeping $\boldsymbol{A}$ on the device between successive matrix-vector products on the device will change the cost of data transfer to O(N) on subsequent iterations, because only the vectors are transferred on these iterations. Figure 7.2 shows the reduction in cost necessary for transfer of only $\boldsymbol{x}$ and $\boldsymbol{y}$ compared to the transfer of $\boldsymbol{A}$, $\boldsymbol{x}$ and $\boldsymbol{y}$.

In an approximation-based optimization environment, finding the optimum solution is an iterative process. The matrices, of order N×M, are only copied to the device once. The subsequent evaluations of (7.1.1) and (7.1.2) account for the significant savings in the computational time. A break-even point can be calculated for each matrix size. For smaller problems, with dimensions of less than 500×500, some routines never reach the break-even point, or require a significant number of iterations to do so. When the benefit of doing the matrix-vector product on the GPU starts to outweigh the cost of data transfer, it is possible to take full advantage of the GPU's computational ability. This derives from the deficiency in Amdahl's law to account for problem size; when the problem dimensions are increased within the device memory capabilities, the performance increases in relation to the available parallel capabilities. Since many approximation-based optimization solvers require large problem sizes, it follows that, for larger

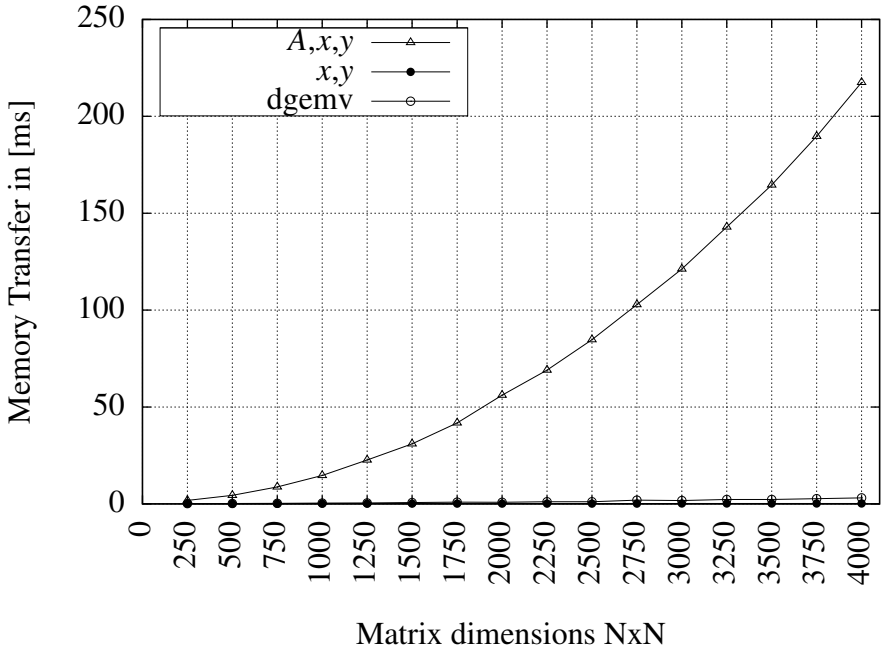


Figure 7.2: The cost of transferring $\boldsymbol{A}$, $\boldsymbol{x}$ and $\boldsymbol{y}$ compared to the transfer of only $\boldsymbol{x}$ and $\boldsymbol{y}$ to and from the device. The matrix-vector product is also shown in order to better compare the transfer to compute cost.

problems, GPU acceleration will be even more beneficial.

7.3.1.1 Performance metrics

The total memory bandwidth M [GB/s] is computed as $M = 10^{-9} \cdot 2(mn + m + n) \times$ sizeof (scalar type)$/T$, where $mn + m + n$ is the input and output scalar count, and T is the execution time or wall clock seconds. The total compute load C [GFlops] will be computed as $C = 10^{-9} \cdot 2m(2n - 1) \cdot /T$, corresponding to m dot products of dimension n (n products and $n - 1$ additions).

7.3.2 Approximation results

Many iterative solution methods in computational science, such as conjugate gradient methods, often reduce to dense linear algebra operations. With the effective performance enhancements of these GPU implementations, dense solution methods become viable for very large systems. This follows from work done by Volkov and Demmel (2008) and Barrachina *et al.* (2008), in which it was demonstrated how to achieve significant percentages of peak floating point throughput and bandwidth on dense matrix operations. Performance enhancements are applied to the Vanderplaats cantilever beam (Vanderplaats, 2001) in a dual SAO environment (see Section 4.4.3).

The performance results in Table 7.2 summarise the performance of each BLAS routine against NetLib BLAS. The problem dimensions are defined by n and $m = n$. k is the number of outer iterations needed to converge to an optimal solution $\boldsymbol{x}^*$ and f^* is the optimal function value. The

function values calculated from the respective routines show negligible change from one routine to another.

To illustrate the required computational orders $O(n^p)$ for each BLAS implementation Figure 7.3 denotes the log-log graph of problem sizes versus CPU-time. The gradient predicts the order of growth for these large dense problems. The orders p are listed in Table 7.1. It is clear that the GPU implementation is superior and that it almost reduces to order $O(n)$ which is similar to that of sparse problems. This is only valid for if problem sizes are kept to within GPU's memory bounds.

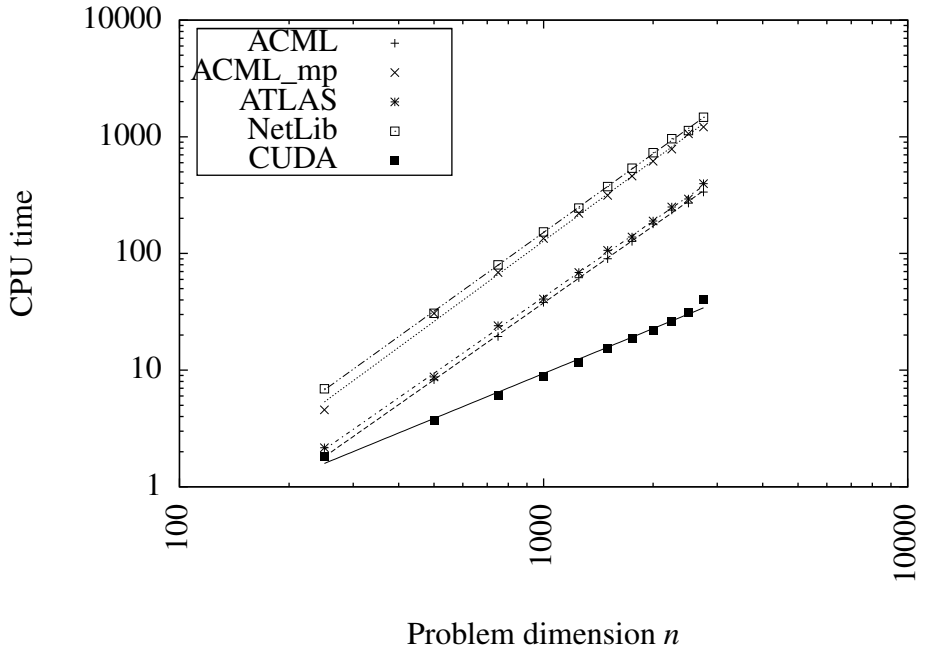


Figure 7.3: Computational order $O(n^p)$ of the respective BLAS implementations.

Table 7.1: Computational order p of the respective BLAS implementations.

BLAS	p
NetLib	2.24
ACML	2.29
ACML_mp	2.17
ATLAS	2.19
CUBLAS	1.28

The results in Figure 7.4 show similar speedup curves for the ACML and ATLAS when compared to the NetLib BLAS implementation. Both the ACML and ATLAS libraries relate approximately the same speedups – around four times that of the NetLib BLAS. As expected,

CUBLAS gives the best performance increase over the NetLib BLAS routine. The ACML_mp matrix-vector multiplication routine shows a speed increase of just over two times that of reference NetLib BLAS. Further tests, not shown, indicate that the performance of ACML_mp decreases when more than two cores are used. It is expected that this performance decrease may be attributed to the limited cache size of the Opteron 275. Problems larger than 2K or 4K read from memory for all processors and therefore do not benefit from the fast cache.

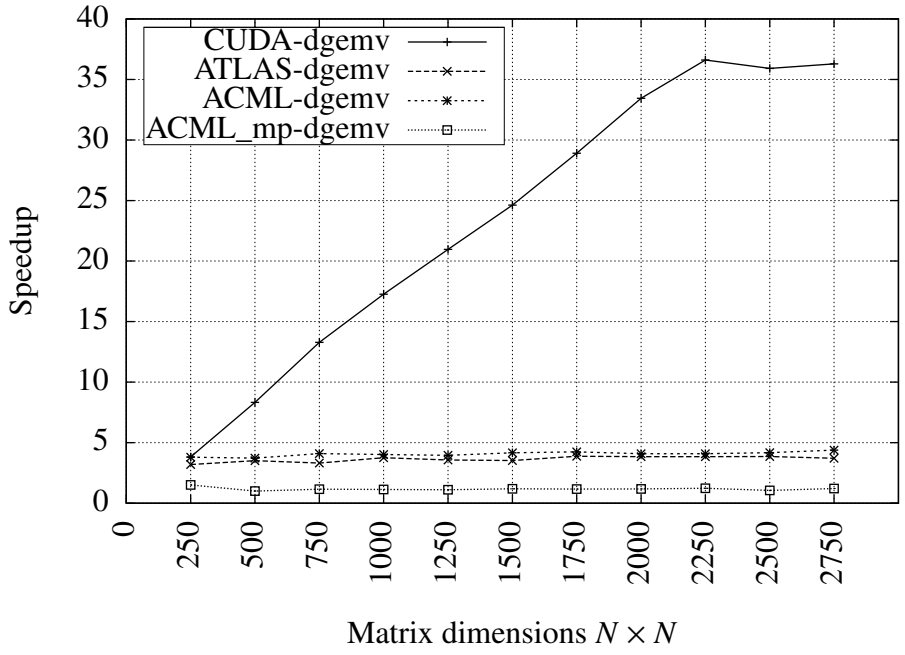


Figure 7.4: SAOi speedups for CUBLAS, ATLAS, ACML and ACML_mp are compared to Netlib BLAS as baseline. The ACML_mp data were collected for two cores.

7.4 Summary

A considerable increase in performance is observed when the computationally expensive `dgemv` routines are moved to the device. Compared to the NetLib BLAS `dgemv` routines, speed increases of up to 36× can be obtained. Speed increases of more than four times can be obtained even for small problems. For platform-optimized routines, these speed increases are lower around 8× depending on the problem size.

No system has unlimited memory, and graphic processing units in particular have strict memory constraints. These devices possess limited on-board memory, with the Geforce GTX 280 limited to 1 GB. This limits our problem dimensions according to

$$\text{global memory} \quad = \quad 2 \times \textbf{\textit{sizeof}}(double) \times n(m+1). \tag{7.4.1}$$

In order to overcome memory constraints, the addition of another Geforce GTX 280 card will enable the problem size to be doubled. This, in turn, will relate to considerably greater perfor-

mance increases by enabling the evaluation of even larger problems by splitting the multiplication across multiple devices.

CUBLAS's `dgemv` normal and transpose matrix-vector multiplication routines have successfully been incorporated into the SAO solver for dense problems in a dual SAO environment. Considerable performance increases have been accomplished in comparison to existing, optimized BLAS routines, as well as multi-threaded BLAS from the ACML_mp libraries. In addition, some benchmarks for pairs of `dgemv` routines where memory access is minimized and data are grouped for continuous once-off memory transfer were provided earlier in this chapter.

Table 7.2: Speedups for the BLAS routines. CPU times are in seconds and the speedups are in reference to NetLiB BLAS in column 4.

			NetLib	CUDA		ATLAS		ACML		ACML_mp	
n	k^*	f^*	CPU	CPU	Speedup	CPU	Speedup	CPU	Speedup	CPU	Speedup
250	11	54067.8	6.92	1.81	3.82	2.16	3.20	1.82	3.80	4.57	1.51
500	11	53891.6	30.75	3.69	8.33	8.75	3.51	8.27	3.72	30.62	1.00
750	11	53832.5	79.75	6.00	13.29	24.05	3.32	19.43	4.10	68.67	1.16
1000	11	53803.0	153.13	8.87	17.26	40.71	3.76	38.12	4.02	134.51	1.14
1250	12	53785.2	245.07	11.69	20.96	68.71	3.57	62.01	3.95	220.75	1.11
1500	12	53773.3	374.44	15.21	24.62	106.39	3.52	90.11	4.16	314.39	1.19
1750	13	53764.8	537.99	18.61	28.91	138.48	3.88	126.75	4.24	460.68	1.17
2000	13	53758.5	731.19	21.86	33.45	190.02	3.85	178.97	4.09	617.29	1.18
2250	13	53753.5	963.31	26.31	36.61	250.47	3.85	235.78	4.09	783.66	1.23
2500	13	53749.6	1129.76	31.46	35.91	292.90	3.86	270.19	4.18	1061.55	1.06
2750	13	53746.3	1470.10	40.51	36.29	396.25	3.71	335.96	4.38	1215.23	1.21

Chapter 8

Conclusion

8.1 Research summary

By combining two of the design tools most widely used by engineers, several possibilities for optimizing SAO codes have been explored.

It was found that the performance of QP solvers in general, and not only the performance between single and double precision, is highly problem dependent. By utilizing single precision solvers, great speed increases may be attained in select cases when the double precision solvers are interchanged with their single precision equivalents.

Next it was found that the use of flagged thread cancellation is detrimental to performance when multiple subsolvers are used in parallel. The problems suffer from unnecessarily long wait times caused by struggling competing routines. However, it was found that, in both cases, the number of outer iterations needed by the parallel algorithm to complete was less than that of the serial algorithms. This has substantial benefits for FEM and/or CFD programs, as many function evaluations are performed between the outer iterations. Therefore, a reduction in the number of outer iterations will ultimately be greatly beneficial to the solution of structural optimization problems.

The parallel multi-start global algorithms proved to be beneficial for computationally demanding constrained and unconstrained cases. The execution times were reduced for all cases. Because of the random nature of the search trajectory, a less than optimal shared memory model was utilized. However, this model ensures convergence because it is not dependent on a set number of processes to evaluate the steps in the search trajectory. This approach makes effective use of the available computation units; however, increasing the amount of CPUs to more than four did not provide any additional performance in our case.

A considerable increase in performance was observed when moving the computationally expensive `dgemv` routines to the device. Compared to NetLib BLAS `dgemv` routines, speed increases of up to 36 times may be reached and 8 times compared to ATLAS BLAS.

8.2 Conclusions

In conclusion, single precision solvers are only beneficial for some problems, and then only for certain types of problems. They are not recommended for the solution of structural optimization problems. In most cases, the use of a different subsolver may be more beneficial than changing to single precision.

The proposed parallel dual-QP method is a very promising alternative when little is known about the problem type in advance. The best solver is always used for the solution. As expected, the dual method remains the most efficient for problems with only a few constraints; however, the QP methods are still superior for solving problems with many design variables and constraints. As the problem properties change at each step, one sees how the preferred solver is selected each time.

Multithreaded parallelization proves to be a simple and effective method to reduce the computational time associated with the solution of both constrained and unconstrained global programming problems. The desired probability of convergence is reached with the least amount of wasted effort, while reducing the effective computational time.

Lastly, CUBLAS's `dgemv` normal and transpose matrix-vector multiplication routines were successfully incorporated into the SAO environment for dense problems. Considerable performance increases may be accomplished in comparison with existing optimized BLAS routines as well as multi-threaded BLAS from the ACML_mp libraries.

8.3 Future work

This thesis has laid the groundwork for much future work. All of these algorithms may be combined, e.g. single precision solvers may be used in parallel with double precision solvers, or double precision solvers may be used in parallel while utilizing GPU acceleration for linear algebra routines.

An in-depth investigation into the strange jump in the performance for the single precision n-variate cantilever problem in Section 4.4.4 must still be done. This was delayed due to time constraints.

Of much consequence is the future work on asynchronous thread cancellation. This forced cancellation will make the parallel solution of subproblems very attractive, because the solvers will be totally independent and no incremental checking of flags will be necessary. Hence, no wait time will be incurred by checking partner threads.

Several additions and optimizations could be made to the GPU algorithms. One would be to distribute the computations over many different devices. The separable quadratic approximations can simply be optimized by moving each `dgemv` pair in the approximation to an individual device, or by distributing all routines to individual devices. This will also alleviate memory restrictions due to memory limitations on an individual device.

Yet another critical field of research of particular importance, especially in the computational sciences, would be Sparse Matrix-Vector Multiplication (SpMV) implementations. These methods

have been utilized effectively in numerous computational disciplines and represent the dominant cost in many iterative methods for solving very large-scale systems. Bell and Garland (2009) provide more information on efficient implementations of sparse matrix-vector multiplication (SpMV) in CUDA.

The development of similar dgemv and SpMV routines in OpenCL by Munshi *et al.* (2009), aiming to increase portability and cross-platform compatibility, is another field of research being investigated. Some optimized GPU-based routines have already been developed and the reader is referred to ACML-GPU (Advanced Micro Devices, Inc., 2010*a*). However these routines do not offer the versatility and memory management capabilities of CUBLAS.

Lastly, generating equivalent OpenCL versions of these `dgemv` and `SpMV` routines, and benchmarking them on both Nvidia and AMD devices, will allow for a more direct comparison between the two competing technologies.

List of References

Advanced Micro Devices, Inc. (2010*a*). *AMD Core Math Library for Graphic Processors*. Available at: `http://developer.amd.com/libraries`

Advanced Micro Devices, Inc. (2010*b*). Introduction to OpenCL™ Programming [Online]. Available at: `http://developer.amd.com/zones/OpenCLZone/courses`, [2010, July].

Amdahl, G.M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In: *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pp. 483–485. ACM, New York, NY.

Arora, J. (1990). Global optimization methods for engineering design. In: *Proc. 31th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*. Long Beach, CA.

Baboulin, M., Buttari, A., Dongarra, J., Kurzak, J., Langou, J., Langou, J., Luszczek, P. and Tomov, S. (2008). Accelerating scientific computations with mixed precision algorithms. *Computing Research Repository*, vol. 180, pp. 2526–2533.

Bainville, E. (2010). Gpu matrix-vector product (gemv). Available at: `http://www.bealto.com/gpu-gemv.html`, [2010, Feb].

Barney, B. (2008). POSIX Threads Programming. *Search*, pp. 1–21. Available at: `https://computing.llnl.gov/tutorials/pthreads/`

Barrachina, S., Castillo, M., Igual, F.D., Mayo, R. and Quintana-ortÃ, E.S. (2008). Solving dense linear systems on graphics processors. In: *14th Int'l European Conference on Parallel Processing*, vol. 5168, pp. 739–748.

Bell, N. and Garland, M. (2009). Efficient sparse matrix-vector multiplication on CUDA. *Proceedings of the 2009 ACM/IEEE conference on Supercomputing*.

Blaauw, G.A. and Brooks, Jr., F.P. (1997). *Computer Architecture: Concepts and Evolution*. 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA. ISBN 0201105578.

Boggan, S. and Pressel, D.M. (2007 August). GPUs: An Emerging Platform for General-Purpose Computation. *U.S. Army Research Lab*. ARL-SR-154.

Bolton, H., Schutte, J. and Groenwold, A. (2000 September). Multiple parallel local searches in global optimization. In: Dongarra, J., Kacsuk, P. and Podhorszki, N. (eds.), *Recent advances in parallel virtual machine and message passing interface*, no. 1908 in Lecture notes in computer science, pp. 88–95. Balatonfüred, Hungary. ISBN 978-3-540-41010-2.

Buttari, A. and Dongarra, J. (2007). Mixed precision iterative refinement techniques for the solution of dense linear systems. *International Journal of High Performance Computing Applications*, vol. 21, no. 4, pp. 457–466.

Buttari, A., Dongarra, J., Kurzak, J., Luszczek, P. and Tomov, S. (2008). Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Transactions on Mathematical Software*, vol. 34, pp. 1–22.

Byrd, R., Lu, P., Nocedal, J. and Zhu, C. (1995). A limited memory algorithm for bound constrained optimization. *SIAM Journal of Scientific Computing*, vol. 16, pp. 1190–1208.

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J. and Menon, R. (2001). *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA. ISBN 1-55860-671-8.

Chapman, B., Jost, G. and Pas, R.V.D. (2008). *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press. ISBN 9780262533027.

Culler, D., Singh, J.P. and Gupta, A. (1999). *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufman. ISBN 1558603433.

Dedu, E., Vialle, S. and Timsit, C. (2000 May). Comparison of OpenMP and classical multithreading parallelization for regular and irregular algorithms. In: Fouchal, H. and Lee, R.Y. (eds.), *Software Engineering Applied to Networking & Parallel/Distributed Computing (SNPD)*, pp. 53–60. Association for Computer and Information Science, Reims, France.

Demmel, J.W. (1997). *Applied numerical linear algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA. ISBN 0-89871-389-7.

Dennis, J.B. and Horn, E.C.V. (1966). Programming semantics for multiprogrammed computations. *Communications of the ACM*, vol. 9, no. 3, pp. 143–155. ISSN 00010782.

Dolan, E.D., More, J.J. and Munson, T.S. (2004 November). Benchmarking optimization software with COPS. Tech. Report ANL/MCS-246, Mathematics and Computer Science Division, Argonne National Laboratory, IL.

Dongarra, J., Croz, J.D. and Hammarling, S. (1988). An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, vol. 14, pp. 1–17.

Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L. and White, A. (2003). *Sourcebook of Parallel Computing*, chap. Part III, p. 852. Morgan Kaufmann Publishers. ISBN 1558608710.

Einarsson, B. (ed.) (2005). *Accuracy and Reliability in Scientific Computing*. Society for Industrial and Applied Mathematics, Philadephia, PA. ISBN 9780898718157.

Etman, L., Groenwold, A. and Rooda, J. (2009 June). On diagonal QP subproblems for sequential approximate optimization. In: *Proc. Eighth World Congress on Structural and Multidisciplinary Optimization*. Lisboa, Portugal. Paper 1065.

Falk, J. (1967). Lagrange multipliers and nonlinear programming. *Journal of Mathematical Analysis and Applications.*, vol. 19, pp. 141–159.

Foster, I. and Kesselman, C. (1997). Globus: A Metacomputing Infrastructure Toolkit. *International Journal of High Performance Computing Applications*, vol. 11, no. 2, pp. 115–128. ISSN 10943420.

Gill, P.E. and Murray, W. (1978). Numerically stable methods for quadratic programming. *Mathematical Programming*, vol. 14, pp. 349–372. ISSN 0025-5610.

Gill, P.E., Murray, W. and Saunders, M.A. (1995). User's guide for qpopt 1.0: A fortran package for quadratic programming. Tech. Rep., Stanford University.

Göddeke, D., Strzodka, R. and Turek, S. (2007 January). Performance and accuracy of hardware-oriented native-, emulated and mixed-precision solvers in fem simulations. *International Journal of Parallel, Emergent and Distributed Systems*, vol. 22, pp. 221–256. ISSN 1744-5760.

Gould, N., Orban, D. and Toint, P.L. (2004). GALAHAD, a library of thread-safe Fortran 90 packages for large-scale nonlinear optimization. *ACM Transactions on Mathematical Software (TOMS)*, vol. 29, pp. 353–372.

Groenwold, A. (2008 November). Personal Interview. Stellenbosch.

Groenwold, A. and Etman, L. (2008). Sequential approximate optimization using dual subproblems based on incomplete series expansions. *Structural Multidisciplinary Optimization*, vol. 36, pp. 547–570.

Groenwold, A. and Etman, L. (2010). *The 'Not-So-Short' manual for the SAOi algorithm*.

Groenwold, A., Etman, L., Snyman, J. and Rooda, J. (2007). Incomplete series expansion for function approximation. *Structural Multidisciplinary Optimization*, vol. 34, pp. 21–40.

Groenwold, A., Etman, L. and Wood, D. (2010). Approximated approximations for SAO. *Structural Multidisciplinary Optimization*, vol. 41, pp. 39–56.

Groenwold, A., Wood, D., Etman, L. and Tosserams, S. (2009). Globally convergent optimization algorithm using conservative convex separable diagonal quadratic approximations. *AIAA Journal*, vol. 47, pp. 2649–2657.

Gropp, W., Lusk, E. and Skjellum, A. (1999*a*). *Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface (Scientific and Engineering Computation)*. The MIT Press. ISBN 0262571323.

Gropp, W., Lusk, E. and Thakur, R. (1999*b*). *Using MPI-2: Advanced Features of the Message Passing Interface*. MIT Press. ISBN 0262571331.

Guerraoui, R. (2010). Foundations of speculative distributed computing - (invited lecture extended abstract). In: *Workshop on Distributed Algorithms/International Symposium on Distributed Computing*, pp. 204–205.

Haftka, R. and Gürdal, Z. (1992). *Elements of Structural Optimization*. Kluwer Academic Publishers, Dordrecht.

Hennessy, J.L. and Patterson, D.A. (2003). *Computer Architecture: A Quantitative Approach*. 3rd edn. Morgan Kaufmann Publishers Inc., San Francisco, CA. ISBN 1558607242.

Higham, N.J. (2002). *Accuracy and Stability of Numerical Algorithms*. 2nd edn. Society for Industrial and Applied Mathematics, Philadelphia, PA. ISBN 0898715210.

Hisley, D., Agrawal, G., Satya-narayana, P. and Pollock, L. (2000). Porting and performance evaluation of irregular codes using openmp. *Concurrency: Practice and Experience*, vol. 12, no. 12, pp. 1241–1259. ISSN 1096-9128.

Huckle, T. (2002). Collection of software bugs. Available at: `http://www5.in.tum.de/~huckle/bugse.html`, [2010, August].

IEEE Standard for Floating-Point Arithmetic (2008). 754-2008 IEEE Standard for Floating-Point Arithmetic.
Available at: `http://ieeexplore.ieee.org`

Khronos Group [Online] (2009). OpenCL. Available at: `http://www.khronos.org/opencl`.

Kirk, D. and Hwu, W.-M. (2010). *Programming massively parallel processors: a hands-on approach*. Elsevier Inc., Morgan Kaufmann Publishers, Burlington, MA.

Langou, J., Langou, J., Luszczek, P., Kurzak, J., Buttari, A. and Dongarra, J. (2006). Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). In: *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, p. 113. ACM, New York, NY. ISBN 0-7695-2700-0.

Laurie, J.F. (2004). Intel Halts Development Of 2 New Microprocessors.
Available at: `http://www.nytimes.com/2004/05/08/business/08chip.html`

Lawson, C.L., Hanson, R.J., Kincaid, D.R. and Krogh, F.T. (1979). Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323.

Lindholm, E., Nickolls, J., Oberman, S. and Montrym, J. (2008). NVIDIA Tesla: A unified graphics and computing architecture. *Micro, IEEE*, vol. 28, no. 2, pp. 39–55.

Macready, D. and Wolpert, W.G. (1997). Performance and Accuracy of Hardware-oriented Native-, Emulated-and Mixed-precision Solvers in FEM Simulations:(Part 2: Double Precision GPUs). *IEEE Transactions on Evolutionary Computation*, vol. 67, no. 1.

Mattson, T.G. (2003 April). How good is OpenMP. *Scientific Programming*, vol. 11, pp. 81–93. ISSN 1058-9244.

Message Passing Interface Forum (1997). MPI-2: Extensions to the Message Passing Interface. Available at: `http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html`

Moore, G.E. (1965). Cramming more components onto integrated circuits. *Electronics*, vol. 38, no. 8, pp. 33–35. ISSN 10984232.

MPI Forum (1994). MPI: A Message-Passing Interface Standard. *Forum American Bar Association*, vol. 8, no. 3/4, pp. 1–299.
Available at: `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`

Mulkay, E.L. and Rao, S.S. (1998). Fuzzy heuristics for sequential linear programming. *Journal of Mechanical Design*, vol. 120, no. 1, pp. 17–23.

Munshi, A. *et al.* (2009). The OpenCL specification version 1.0. *Khronos OpenCL Working Group*.

NAG Library Manual, Mark 22 (2009). *E04NQF NAG Library Manual*. The Numerical Algorithms Group Ltd, Oxford.
Available at: `http://www.nag.co.uk/numeric/fl/nagdoc_fl22/pdf/E04/e04nqf.pdf`

Nikolopoulos, D.S., Polychronopoulos, C.D. and Ayguadé, E. (2001). Scaling irregular parallel codes with minimal programming effort. In: *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, Supercomputing '01, pp. 16–16. ACM, New York, NY. ISBN 1-58113-293-X.

Nocedal, J. and Wright, S. (2006). *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering, 2nd edn. Springer.

NVIDIA Corporation (2008). *CUDA Programming Guide*. NVIDIA: Santa Clara, CA.
Available at: `http://www.nvidia.com`

NVIDIA Corporation (2009). *NVIDIA OpenCL Best Practices Guide*.
Available at: `http://www.nvidia.com`

NVIDIA Corporation (2010 February). CUBLAS Library. Tech. Rep. PG-00000-002_V3.0, NVIDIA Corporation, Santa Clara, CA.

NVIDIA Corporation (2011). Directcompute [online]. Available at: `http://www.nvidia.com/object/cuda_directcompute.html`, [2011, May].

Omondi, A.R. (1994). *Computer Arithmetic Systems: Algorithms, Architecture and Implementation*. Prentice Hall International Ltd., Hertfordshire, UK. ISBN 0-13-334301-4.

OpenMP 3.0 (2008). OpenMP Application Program Interface. Tech. Rep., Openmp Architecture Review Board.
Available at: `http://www.openmp.org/specs`

Pacheco, P.S. (1997). *Parallel Programming with MPI*. Morgan Kaufmann. ISBN 1558603395.

POSIX (2004). *IEEE Standard for Information Technology: Portable Operating System Interface (POSIX) Base Definitions*. IEEE Computer Society Press. IEEE Standard.

Rabaey, J.M. (1996). *Digital Integrated Circuits: A Design Perspective*. Prentice-Hall, Inc., Upper Saddle River, NJ. ISBN 0-13-178609-1.

Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W. and Dongarra, J. (1998). *MPI – The Complete Reference: Volume 1*. 2nd edn. MIT Press. The MPI Core.

Snyman, J., Bolton, H. and Groenwold, A. (2003 June). A multi-start methodology for global optimization using novel local constrained optimizers. In: Floudas, C. and Pardalos, P. (eds.), *Proc. 4th International Conference on Frontiers in Global Optimization*. Santorini, Greece.

Snyman, J., Bolton, H. and Groenwold, A. (2004). A multi-start methodology for global optimization using novel constrained local optimizers. In: Floudas, C. and Pardalos, P. (eds.), *Frontiers In Global Optimization*, vol. 74 of *Nonconvex Optimization and It's Applications*, pp. 499–516. Kluwer Academic Publishers. ISBN 978-1-402-07699-2.

Snyman, J. and Fatti, L. (1987). A multi-start global minimization algorithm with dynamic search trajectories. *Journal of Optimization Theory and Applications*, vol. 54, pp. 121–141.

Stevenson, D.e. (1999 May). A critical look at quality in large-scale simulations. *Computing in Science and Engg.*, vol. 1, pp. 53–63. ISSN 1521-9615.

Strzodka, R. and Göddeke, D. (2006*a*). Mixed precision methods for convergent iterative schemes. In: *Proceedings of the Workshop on Edge Computing Using New Commodity Architectures*, vol. EDGE 2006, pp. 23.–24.

Strzodka, R. and Göddeke, D. (2006*b*). Pipelined mixed precision algorithms on FPGAs for fast and accurate PDE solvers from low precision components. In: *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 259–268.

Süß, M. and Leopold, C. (2006*a*). Common mistakes in OpenMP and how to avoid them. In: *Proceedings of the International Workshop on OpenMP - IWOMP'06*. Wilhelmshöher Allee 73, D-34121 Kassel, Germany.

Süß, M. and Leopold, C. (2006*b*). Implementing irregular parallel algorithms with OpenMP what's missing and how to solve it. Tech. Rep., University of Kassel, Research Group Programming Languages / Methodologies, Wilhelmshöher Allee 73, D-34121 Kassel, Germany.

Svanberg, K. (1987). The method of moving asymptotes - a new method for structural optimization. *International Journal for Numerical Methods in Engineering*, vol. 24, pp. 359–373.

Svanberg, K. (2002). A class of globally convergent optimization methods based on conservative convex separable approximations. *SIAM Journal on Optimization*, vol. 12, pp. 555–573.

Tweakers.net (2007). Chip magicians at work: patching at 45nm.
Available at: `http://tweakers.net/reviews/`

Vanderplaats, G. (2001). *Numerical optimization techniques for engineering design*. Vanderplaats R&D, Inc., Collorado Springs.

Volkov, V. and Demmel, J.W. (2008). Benchmarking gpus to tune dense linear algebra. In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pp. 31:1–31:11. IEEE Press, Piscataway, NJ. ISBN 978-1-4244-2835-9.

Whaley, R. and Dongarra, J. (1998). Automatically tuned linear algebra software. *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*.

Wilkinson, J.H. (1960). Error analysis of floating-point computation. *Numerische Mathematik*, vol. 2, pp. 319–340. ISSN 0029-599X.

Wood, D., Groenwold, A. and Etman, L. (2010). Bounding the dual of Falk to circumvent the requirement of relaxation in globally convergent SAO algorithms. *Optimization and Engineering*. Provisionally accepted.

Wrench, J.W. (1970). Table errata: *The art of computer programming, Vol. 2: Seminumerical algorithms* (Addison-Wesley, Reading, Mass., 1969) by Donald E. Knuth. *Mathematics of Computation*, vol. 24.

Ypma, T.J. (1995). Historical development of the newton-raphson method. *Siam Review*, vol. 37.

Zhang, Y. (1994). On the convergence of a class of infeasible interior-point methods for the horizontal linear complementarity problem. *SIAM Journal on Optimization*, vol. 4, no. 1, pp. 208–227.

Zhu, C., Byrd, R., Lu, P. and Nocedal, J. (1994). L-BFGS-B: FORTRAN subroutines for large scale bound constrained optimization. Tech. Rep. NAM-11, Northwestern University, EECS Department.

Zielinsky, R. (1981). A statistical estimate of the structure of multiextremal problems. *Mathematical Programming*, vol. 21, pp. 348–356.

Appendix A

Computational accuracy

Numerical software is used in engineering to design cars, aeroplanes, boats, bridges, power plants and refineries. Because of the financial and human stakes involved, it is essential that design software be accurate, reliable and robust. A brief history of precision and accuracy is provided in this appendix. We further describe the difficulties in working with floating-point values and how they affect scientific computing according to the IEEE 754 standard.

A.1 Floating-point numbers and IEEE arithmetic

The precision of any value in scientific computing describes the number of significant digits that are used to express that value. On all computational devices, a result in floating-point arithmetic is rounded to a given or fixed precision, which is the length of the resulting significand[1]. For the last ten years the most commonly encountered representation is that defined by the IEEE 754 Standard (IEEE Standard for Floating-Point Arithmetic, 2008). The standard defines four basic formats, named using their base and the number of bits used to encode them. These are single, extended single, double and extended double precision. A *floating-point number*, x, can be represented as

$$x = \pm m \cdot b^{e-t}, \tag{A.1.1}$$

where b is the *base* or *radix*, t is the *precision*, e is the *exponent*, with an exponent range of $[e_{\min}, e_{\max}]$, and m is the *significand*. The significand must satisfy $0 \leq m \geq b^t - 1$ (Einarsson, 2005).

A.1.1 IEEE Standard for Binary Floating Point Arithmetic

The IEEE (Institute of Electrical and Electronics Engineers) standard for floating point arithmetic specifies how single precision (32 bit) and double precision (64 bit) floating-point numbers

[1] Derived from the quasi-logarithmic nature of floating-point representations, *mantissa* has always been used to describe the significand. However, according to Wrench (1970) and Blaauw and Brooks (1997), this is incorrect. Significand is the term used in the IEEE standard.

are to be represented, as well as how arithmetic should be carried out on them. The differences in the formats affect the accuracy and data rate of floating-point computations. It is these differences that we wish to exploit.

We will only discuss the two main floating-point formats: single and double precision. The IEEE standard does includes many more, such as extended precision, format conversion operations and other operations (add, subtract, divide, remainder and square root) that require rounding during execution. The four rounding modes are: round down, round up, round toward zero and round to nearest. For an in-depth discussion of floating-point arithmetic, the reader is referred to Omondi (1994, Part II)

The IEEE single precision floating-point standard representation requires a 32-bit word, which may be represented as numbered from 0 to 31, from left to right. The first bit is the sign bit, s, the next eight bits are the exponent bits, e, and the remaining 23 bits are the fractional part f of the significand m:

```
S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF
0 1       8 9                     31
```

The value x represented by the word may be determined as follows:

1. If $e = 255$ and f is nonzero, then $x =$ NaN ("Not a number").
2. If $e = 255$ and f is zero and s is 1, then $x = -\infty$.
3. If $e = 255$ and f is zero and s is 0, then $x = \infty$.
4. If $0 < e < 255$, then $x = (-1)^s \cdot 2^{e-127} \cdot (1.f)$, where $1.f$ is intended to represent the binary number created by prefixing f with an implicit leading 1 and a binary point.
5. If $e = 0$ and f is nonzero, then $x = (-1)^s \cdot 2^{-126} \cdot 0.f$. These are "unnormalized" values.
6. If $e = 0$ and f is zero and s is 1, then $x = -0$.
7. If $e = 0$ and f is zero and s is 0, then $x = 0$.

The IEEE double precision-floating point standard representation requires a 64-bit word, which may be represented as numbered from 0 to 63, from left to right. The first bit is the sign bit, s, the next eleven bits are the exponent bits, e, and the remaining 52 bits are the fraction f:

```
S EEEEEEEEEEE FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
0 1          11 12                                                63
```

The value x represented by the word may be determined as follows:

1. If $e = 2047$ and f is nonzero, then $x =$ NaN.
2. If $e = 2047$ and f is zero and s is 1, then $x = -\infty$.

3. If $e = 2047$ and f is zero and s is 0, then $x = \infty$.
4. If $0 < e < 2047$, then $x = (-1)^s \cdot 2^{e-1023} \cdot (1.f)$ where $1.f$ is intended to represent the binary number created by prefixing f with an implicit leading 1 and a binary point.
5. If $e = 0$ and f is nonzero, then $x = (-1)^s \cdot 2^(- 1022) \cdot (0.f)$. These are "unnormalized" values.
6. If $e = 0$ and f is zero and s is 1, then $x = -0$.
7. If $e = 0$ and f is zero and s is 0, then $x = 0$.

This description has been adapted from the ANSI/IEEE Standard for Binary Floating Point Arithmetic (IEEE Standard for Floating-Point Arithmetic, 2008).

A.2 Floating-point error analysis

Floating-point precision must be sufficient to attain a correct and reliable result (Einarsson, 2005). There are several examples where numerical problems have not only led to financial loss, but loss of lives. The reader is referred to the interesting paper by Stevenson (1999), and the Thomas Huckle website, containing a collection of software bugs (Huckle, 2002). There are four basic problems with floating point arithmetic: rounding errors, cancellation errors, recursion errors and integer overflow errors (Omondi, 1994).

Rounding errors occur because calculations can only be performed with a fixed number of significant digits, thus, after each operation, the result must be rounded, introducing a rounding error. *Cancellation errors* occur when two almost equal quantities are subtracted. Assume $x_1 = 1.243 \pm 0.0005$ and $x_2 = 1.134 \pm 0.0005$; then $x_1 - x_2 = 0.009 \pm 0.001$, a result in which several significant digits have been lost, resulting in a large relative error (Einarsson, 2005). *Recursion errors*, often referred to as cumulative errors, occur in scientific computing when a new entity needs to be calculated based on a previous one, in either an iterative or recursive process, updating the values all the time. *Integer overflow* occurs when the range of integers is larger than the range of floating-point values.

Verification and validation of computed results is not a simple process. In order to measure the quality of a computed result, we use three gradations (Einarsson, 2005):

- **precision**, the number of significant digits available for input, output and arithmetic;
- **accuracy**, the absolute or relative error of an approximate quantity;
- **reliability**, which measures the percentage of software failure; the true error is greater than the error bound.

Scientific and engineering problems are susceptible to the sensitivity of a problem and the sensitivity of the solution method that is used. The *condition* of a problem is concerned with the sensitivity of the problem and is independent of the solution method. Ill-conditioned problems

are problems in which relatively small changes in the data cause large changes in the solution. The *stability* of a method is concerned with the sensitivity of the method to rounding errors during the solution process. A method is deemed unstable when the solution method does not guarantee a solution as accurate as the data.

Floating-point error analysis is concerned with the analysis of errors in the presence of floating-point arithmetic. To determine the accuracy of floating-point arithmetic we must use *relative error* calculations. Einarsson (2005, Chapter 4) gives a brief introduction to floating-point error analysis, more detailed discussions can be found in Higham (2002) and Wilkinson (1960). In order to verify that our computed results are in fact correct, we test for convergence to some error e of the norm during our optimization loops. This error value must enable single precision (32 bit) codes to converge. Because the problems that we are testing for have no analytic solution we compare the single precision results with the double precision results (hoping that the double precision results are more accurate). Once the norm of the single precision result deviates from that of the double precision result, we know the largest size of our error bound. This is the most effective method without going into a detailed algorithmic and problem analysis. See the results for the respective problems in Chapter 4.

Appendix B

The hardware and software

B.1 The computation machine

Most computations were performed on a 64-bit 3.73 GHz Intel Xeon-based Dell machine with eight cores and 32 GiB of RAM. The machine runs openSUSE 11.3 with the Linux 2.6.34-12-desktop x86_64 kernel and GCC 4.3.2.

The computations for all GPU-enabled tests were obtained on a Dual Opteron 275 system with a total of four CPU cores running at 2.2 GHz and 16 GB of RAM. For the GPU results, a NVIDIA GTX 280 with 1 GB of onboard memory was used. The system runs Ubuntu Linux 2.6.27.37 x86_64 and GCC 4.3.2 with CUDA 3.2.1.

B.2 SAOi

The SAO*i* algorithm by Groenwold and Etman (2010) is a sequential approximate optimization (SAO) algorithm for bounded or inequality constrained nonlinear programming problems based on convex separable approximation functions. These problems are of the form

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathcal{R}^n} \; & f_0(\boldsymbol{x}), && \boldsymbol{x} = [x_1, x_2, ..., x_n]^T \\ \text{subject to} \; & g_j(\boldsymbol{x}) \leq 0, && j = 1, 2, \ldots, m, \\ & \check{x}_i \leq x_i \leq \hat{x}_i, && i = 1, 2, \ldots, n \end{aligned} \tag{B.2.1}$$

where $f_0(\boldsymbol{x})$ represents a real-valued scalar objective function and $g_j(\boldsymbol{x})$ represents m real-valued scalar inequality constraint functions. Both $f_0(\boldsymbol{x})$ and $g_j(\boldsymbol{x})$ depend on n design variables $\boldsymbol{x} = [x_1, x_2, ..., x_n]^T \in \mathcal{R}^n$. The upper and lower bound constraints are represented by $\hat{x}_i$ and $\check{x}_i$ respectively.

The subproblems may be solved in the dual space or via a diagonalized quadratic programming (QP) approach. It is often desirable in structural optimization to construct diagonal quadratic approximations, as follows

$$\tilde{f}(\boldsymbol{x}) = f^{\{k\}} + \boldsymbol{\nabla}^T f^{\{k\}} \boldsymbol{s} + \frac{1}{2} \boldsymbol{s}^T \boldsymbol{C}_j^{\{k\}} \boldsymbol{s}, \tag{B.2.2}$$

$C_j^{(k)}$ in (B.2.2) is the only unknown and is determined by any quadratic, reciprocal, exponential or other approximation. However, the user may select approximate diagonal curvatures that describe the problem best. Since this quadratic approximation accommodates a very simple form of the dual and QP subproblems, the formal link between the SQP and SAO methods is made here (Etman *et al.*, 2009). Therefore, SAOi has the ability to solve subproblems based on duality and QP statements. At the moment the following subsolvers are interfaced: the l-BFGS-b dual solver (Byrd *et al.*, 1995; Zhu *et al.*, 1994), the commercial NAG `E04NQF` QP solver and the range of QP solvers available in `GALAHAD` (Gould *et al.*, 2004), which are freely available to academics.

Modifications have been made to the SAO routine in order to include manual relaxation for the `LSQP GALAHAD` routine. This is necessary because the `LSQP` routines do not include automatic relaxation. For more in-depth information on the SAO*i* algorithm, please see the "not-so-short" manual by Groenwold and Etman (2010).

B.3 `E04NQF` double and single precision solvers

`E04NQF` solves large-scale sparse linear programming or convex quadratic programming problems. These problems have the form of equation (2.1.1). The initialization routine, `E04NPF`, must be called before calling `E04NQF`. The single precision version is a non-standard version of the double precision `E04NQF` subsolver, which was modified specifically for our use by the NAG developers to include a single precision version for our research purposes.

The `E04NQF` algorithm is based on an inertia-controlling method similar to that of Gill and Murray (1978), and is described further in Gill *et al.* (1995). This method comprises two phases: a *feasibility phase*, in which an initial feasible point is found by minimizing the sum of the infeasibilities, and on the *optimality phase*, in which the quadratic objective function is minimized within the feasible region. Both computation phases are completed by the `E04NQF` subroutine.

The `E04NQF` function specification looks as follows:

```
subroutine e04nqf(start, qphx, m, n, ne, nname, lenc, ncolh, iobj,
                  objadd, prob, acol, inda, loca, bl, bu, c, names,
                  helast, hs, x, pi, rc, ns, ninf, sinf, obj, cw, lencw,
                  iw, leniw, rw, lenrw, cuser, iuser, ruser, ifail)

integer           m, n, ne, nname, lenc, ncolh, iobj, inda(ne),
                  loca(n+1), helast(n+m), hs(n+m), ns, ninf, lencw,
                  iw(leniw), leniw, lenrw, iuser(*), ifail

double precision  objadd, acol(ne), bl(n+m), bu(n+m), c(max(1,lenc)),
                  x(n+m), pi(m), rc(n+m), sinf, obj, rw(lenrw), ruser(*)

character*1       start
```

```
character*8       prob, names(nname), cw(lencw), cuser(*)
external          qphx
```

For single precision, we replace:

```
real              objadd, acol(ne), bl(n+m), bu(n+m), c(max(1,lenc)),
                  x(n+m), pi(m), rc(n+m), sinf, obj, rw(lenrw), ruser(*)
```

Before calling E04NQF or one of the option setting routines E04NRF, E04NSF, E04NTF or E04NUF, E04NPF must be called. These functions are all written in the Fortran 95 programming language and support both sparse and dense matrix storage systems. For further information the reader is referred to the E04NQF NAG Library Manual, Mark 22 (2009) library routine document.

B.4 LSQP double and single precision solvers

The LSQP package uses a primal-dual interior-point method to solve large-scale linear or separable convex quadratic programming problems. These problems are also of the form given in equation (2.1.1). The basic algorithm is that of Zhang (1994), with further enhancements made by Gould *et al.* (2004).

There are two versions, GALAHAD_LSQP_single and GALAHAD_LSQP_double. Both these versions use the GALAHAD_SYMBOLS, GALAHAD_QPT, GALAHAD_SMT, GALAHAD_SPECFILE, GALAHAD_QPP, GALAHAD_ROOTS, GALAHAD_SILS and GALAHAD_FDC packages. All functions are written in either Fortran 95, TR 15581 or Fortran 2003 languages and support sparse and dense matrix storage systems. Both the single and double precision versions are proper production code. The LSQP function specification looks as follows for double precision:

```
subroutine galahad_lsqp (n, m, a_ne, f_a, c_l, c_u, x_l, x_u,
                         a_row, a_col, a_ptr, x, g, w, y, z,
                         a_val, string, ierr, ks, f)

use galahad_lsqp_double
implicit none
integer, parameter             :: wp = kind( 1.0d+0 )
real ( kind = wp ), parameter  :: infinity = 10.0_wp ** 20
type ( qpt_problem_type )      :: p
type ( lsqp_data_type )        :: data
type ( lsqp_control_type )     :: control
type ( lsqp_inform_type )      :: info
integer                        :: n, m, a_ne, ks
real ( kind = wp )             :: f, f_a, c_l(m), c_u(m), x_l(n), x_u(n)
real ( kind = wp )             :: x(n), y(m), z(n), a_val(a_ne)
real ( kind = wp )             :: g(n), w(n)
```

```
integer                         :: i, s, ierr
integer                         :: a_row(a_ne), a_col(a_ne), a_ptr(m+1)
character(len=*)                :: string
```

Access to the package requires a USE statement, such as the single precision version USE GALAHAD _LSQP_single and the double precision version USE GALAHAD_LSQP_double. If it is required that both modules should be used at the same time, the derived types SMT_type, QPT_problem_type, LSQP_time_type, LSQP_control_type, LSQP_inform_type and LSQP_data_type and the subroutines LSQP_ initialize, LSQP_solve, LSQP_terminate and LSQP_read_specfile, must be renamed in one of the USE statements. The precision must also be changed. KIND(8) needs to be replaced with KIND(4) for single precision. For more information the reader is referred to the GALAHAD LSQP package specification (Gould *et al.*, 2004).

B.5 l-BFGS-b dual solver

The dual solver is a limited memory, bound constrained solver for solving large nonlinear optimization problems. It is intended for problems in which information on the Hessian matrix is difficult to obtain, or for large dense problems. The algorithm is presented in Byrd *et al.* (1995), and the FORTRAN solver was developed by Zhu and co-workers (Zhu *et al.*, 1994).

The algorithm is implemented in Fortran 77, in double precision. In order to allow the user complete control over these computations, reverse communication is used. The routine lbfgsb.f must be called repeatedly under the control of the variable task. The calling statement of l-BFGS-b is

```
call lbfgsb(n,m,x,l,u,nbd,f,g,factr,wa,iwa,task,iprint,isbmin,
            csave,lsave,isave,dsave),
```

and the l-BFGS-b function specification looks as follows:

```
subroutine lbfgsb (n, m, x, l, u, nbd, f, g, factr, wa, iwa, task,
                   iprint, isbmin, csave, lsave, isave, dsave)

implicit none
integer          umax
parameter        (umax = 8)
character*60     task, csave
logical          lsave(4)
integer          n, m, iprint ,nbd(m), iwa(3*ni), isave(44) , isbmin
double precision l(m), u(m), g(m), f, dsave(29), x(n), factr
double precision wa(2*umax*ni + 4*ni + 11*umax*umax + 8*umax)
```

Appendix C

Proof of the Bayesian stopping criterion

We present here an outline of the stopping criterion used, and closely follow the presentation in Snyman and Fatti (1987). (It has since been learned that the proof may be shown to be a generalization of the procedure proposed by Zielinsky (Zielinsky, 1981).)

C.1 The stopping criterion

Let r be the number of sample points falling within the region of convergence of the current overall minimum $\tilde{f}$ after $\tilde{n}$ points have been sampled. Then, the probability that $\tilde{f}$ be equal to f^* satisfies

$$\Pr[\tilde{f} = f^*] \geq q(\tilde{n}, r) = 1 - \frac{(\tilde{n}+1)!\,(2\tilde{n}-r)!}{(2\tilde{n}+1)!\,(\tilde{n}-r)!}$$

Proof:

Given $\tilde{n}^*$ and α^*, the probability that at least one point, $\tilde{n}^* \geq 1$, has converged to f^* is

$$\Pr[\tilde{n}^* \geq 1|\tilde{n}, r] = 1 - (1-\alpha^*)^{\tilde{n}}\ . \tag{C.1.1}$$

In the Bayesian approach, we characterize our uncertainty about the value of α^* by specifying a prior probability distribution for it. This distribution is modified using the sample information (namely $\tilde{n}$ and r) to form a posterior probability distribution. Let $p_*(\alpha^*|\tilde{n}, r)$ be the posterior probability distribution of α^*. Then,

$$\begin{aligned}\Pr[\tilde{n}^* \geq 1|\tilde{n}, r] &= \int_0^1 \left[1-(1-\alpha^*)^{\tilde{n}}\right] p_*(\alpha^*|\tilde{n}, r)d\alpha^* \\ &= 1 - \int_0^1 (1-\alpha^*)^{\tilde{n}} p_*(\alpha^*|\tilde{n}, r)d\alpha^*. \end{aligned}\tag{C.1.2}$$

Now, although the r sample points converge to the current overall minimum, we do not know whether this minimum corresponds to the global minimum of f^*. We proceed as follows:

Let R_k denote the region of convergence of local minimum $\hat{\boldsymbol{x}}^k$, and let α_k be the associated probability that a sample point be selected in R_k. The region of convergence and the associated

probability for the global minimum $\boldsymbol{x}^*$ are denoted by R^* and α^* respectively. The following basic assumption, which is probably true for many functions of practical interest, is now made.

Basic assumption:

$$\alpha^* \geq \alpha_k \text{ for all local minima } \hat{\boldsymbol{x}}^k. \tag{C.1.3}$$

Noting that $(1-\alpha)^{\tilde{n}}$ is a decreasing function of α, the replacement of α^* in (C.1.2) by α yields

$$\Pr[\tilde{n}^* \geq 1|\tilde{n}, r] \geq \int_0^1 \left[1-(1-\alpha)^{\tilde{n}}\right] p(\alpha|\tilde{n}, r)d\alpha \ . \tag{C.1.4}$$

Now, using Bayes' theorem we obtain

$$p(\alpha|\tilde{n}, r) = \frac{p(r|\alpha, \tilde{n})p(\alpha)}{\int_0^1 p(r|\alpha, \tilde{n})p(\alpha)d\alpha} \ . \tag{C.1.5}$$

Since the $\tilde{n}$ points are sampled at random and each point has a probability α of converging to the current overall minimum, r has a binomial distribution, with parameters α and $\tilde{n}$. Therefore

$$p(r|\alpha, \tilde{n}) = \binom{\tilde{n}}{r}\alpha^r(1-\alpha)^{\tilde{n}-r} \ . \tag{C.1.6}$$

Substituting (C.1.6) and (C.1.5) into (C.1.4) gives:

$$\Pr[\tilde{n}^* \geq 1|\tilde{n}, r] \geq 1 - \frac{\int_0^1 \alpha^r(1-\alpha)^{2\tilde{n}-r}p(\alpha)d\alpha}{\int_0^1 \alpha^r(1-\alpha)^{\tilde{n}-r}p(\alpha)d\alpha} \ . \tag{C.1.7}$$

A suitable, flexible prior distribution, $p(\alpha)$ for α is the beta distribution with parameters a and b. Hence,

$$p(\alpha) = [1/\boldsymbol{\beta}(a,b)]\,\alpha^{a-1}(1-\alpha)^{b-1}, \qquad 0 \leq \alpha \leq 1. \tag{C.1.8}$$

Using this prior distribution gives:

$$\begin{aligned}\Pr[\tilde{n}^* \geq 1|\tilde{n}, r] \ &\geq \ 1 - \frac{\Gamma(\tilde{n}+a+b)\,\Gamma(2\tilde{n}-r+b)}{\Gamma(2\tilde{n}+a+b)\,\Gamma(\tilde{n}-r+b)} \\ &= \ 1 - \frac{(\tilde{n}+a+b-1)!\,(2\tilde{n}-r+b-1)!}{(2\tilde{n}+a+b-1)!\,(\tilde{n}-r+b-1)!},\end{aligned}$$

Assuming a prior expectation of 1, (viz. $a = b = 1$), we obtain

$$\Pr[\tilde{n}^* \geq 1|\tilde{n}, r] \ \geq \ q(\tilde{n}, r) \ = \ 1 - \frac{(\tilde{n}+1)!\,(2\tilde{n}-r)!}{(2\tilde{n}+1)!\,(\tilde{n}-r)!},$$

which is the required result.

In practice, the Stopping Rule becomes: given a prescribed target probability q^*, stop when $q(\tilde{n}, r) \geq q^*$.

Appendix D

OpenCL dgemv

This abstract will see an OpenCL-mv implementation of the general matrix-vector product. This function is known in the BLAS standard library as dgemv (double precision).

D.1 OpenCL matrix-vector product (gemv)

The aim is to solve the function known in the BLAS standard library as sgemv (single precision) and dgemv (double precision). The given inputs are an $m \times n$ matrix $\boldsymbol{A}$ ($\boldsymbol{A}$ has m rows and n columns) and an input vector $\boldsymbol{x}$ of dimension n. The solution vector $\boldsymbol{y}$ can then be computed and is of dimension m. This is a reduced version of the NetLib BLAS routine which is written as $\boldsymbol{y} = \alpha\boldsymbol{A}\boldsymbol{x} + \beta\boldsymbol{y}$.

For factor values $\alpha = 1$ and $\beta = 0$, this routine reduces to $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$, which is what will be implemented in our `dgemv` routine using OpenCL. The product therefore may be defined as

$$\boldsymbol{y}_i = \sum_{i=1}^{m} \boldsymbol{A}_{ik} \cdot \boldsymbol{x}_k. \tag{D.1.1}$$

Component i of $\boldsymbol{y}$ is the dot product of row i of $\boldsymbol{A}$ and vector $\boldsymbol{x}$, both vectors of dimension n.

Matrix $\boldsymbol{A}$ is stored in column-major order. In memory, the matrix is represented by an array of $m \times n$ scalars (float or double). The first m values are the first column of $\boldsymbol{A}$, the next m values are the second column, etc. Vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are represented by arrays of n and m contiguous values respectively.

D.2 Implementation

Bainville (2010) describes three OpenCL implementations of the general matrix-vector product. The first version calculates one component of $\boldsymbol{y}$ per thread. The second initializes p threads, each calculating a component of $\boldsymbol{y}$, which is then stored in local memory and reduced in the end. The third version applies two consecutive kernels, each calculating p threads per component of

$\boldsymbol{y}$. The first kernel preloads the corresponding portion of $\boldsymbol{x}$ in local memory to make it available to all threads of the group, then stores the output in an $m \times p$ matrix in global memory to be reduced by the second kernel. The third implementation is the most efficient, therefore we will focus on this implementation.

In this version, the requirements in local memory for each group are reduced to n/p scalars. This allows more workgroups to run concurrently in each stream processor. Thus, when the value of p is small enough, the cost of the reduction becomes negligible compared to the computation of all partial sums. Another advantage is the reuse of local memory; by copying the $\boldsymbol{x}$ slices of local memory, the work group width now resides on one thread only. Therefore, all threads are not confined to the same row to communicate and the total work group capacity can be assigned to the computation of rows (Bainville, 2010).

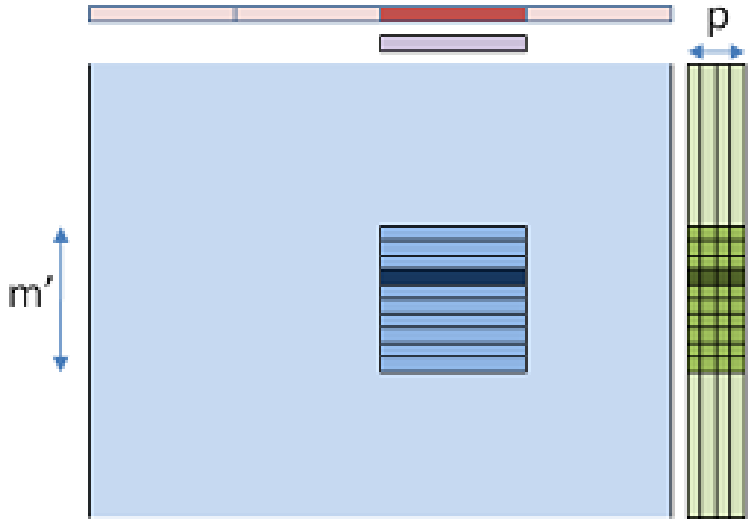


Figure D.1: A work group (medium blue) contains m' by 1 threads. Each group first copies one slice of $\boldsymbol{x}$ in local memory (pink), then each thread computes its partial dot product (dark blue) and stores it in one cell of $\boldsymbol{y}$ (green), which has been extended to p columns. The columns of $\boldsymbol{y}$ are added together using a second kernel. (Bainville, 2010).

D.3 Benchmarks

The benchmarks in Figure D.2 are for two successive matrix-vector products. This test evaluates only the quadratic approximation of equation (7.1.1). Note that the cost of data transfer to the device is not accounted for during the calls to the matrix-vector multiplications on the device. Figure D.2 shows GFLOPS for each matrix-vector pair for increasing problem sizes. The problem sizes are padded to fit the thread limits for each workgroup on the GPU.

From Figure D.2 it is clear that the OpenCL matrix-vector multiplication routine is superior if memory transfer is not taken into account. However, compared to the CUBLAS memory copy

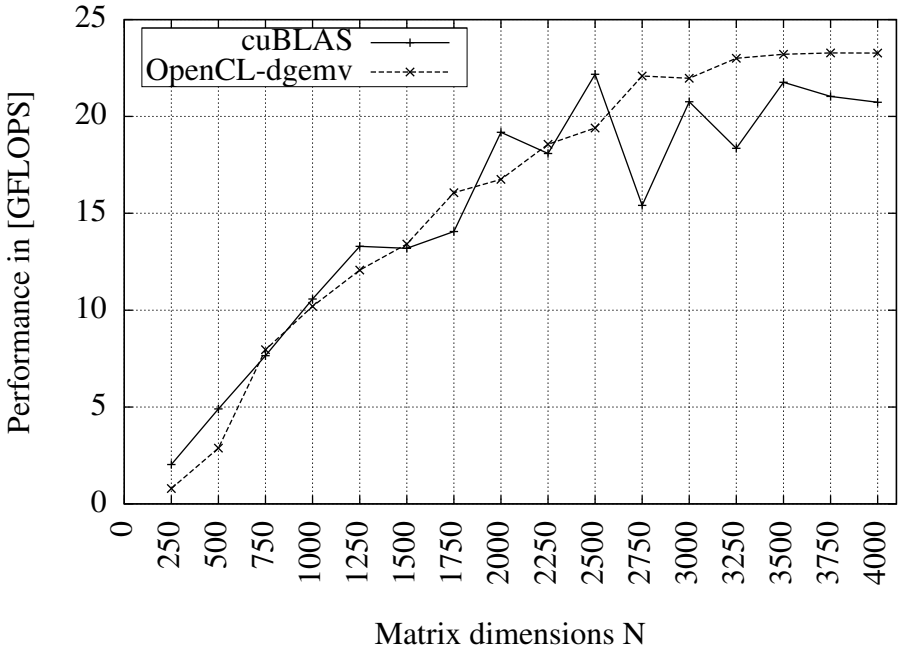


Figure D.2: The performance in GFLOPS for two successive OpenCL matrix-vector and CUBLAS calls.

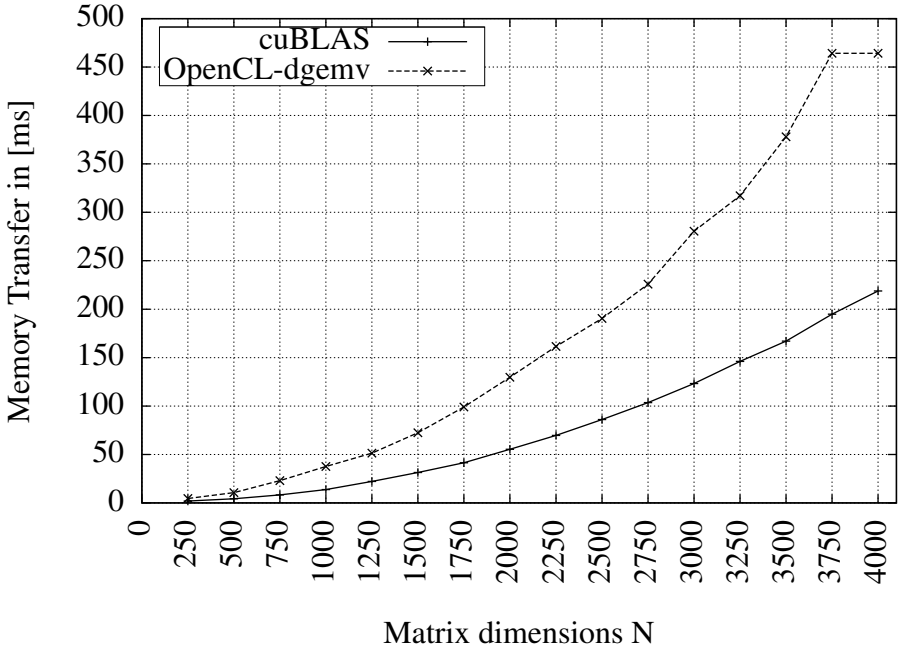


Figure D.3: The memory copies for two successive OpenCL matrix-vector and CUBLAS calls.

routine, the OpenCL routine is lacking in performance. This is a further optimization that needs to be done on this implementation.

D.4 Results

The `dgemv` routine was applied to the Vanderplaats cantilever beam (Vanderplaats, 2001) within the dual SAO environment (see Section 4.4.3).

The performance results for the OpenCL routine are weighted against those of the NetLib BLAS library. The problem dimensions are defined by n and m, and k is the number of outer iterations needed to converge to $\boldsymbol{x}^*$. f^* is the optimal function value and h^* is the relative error.

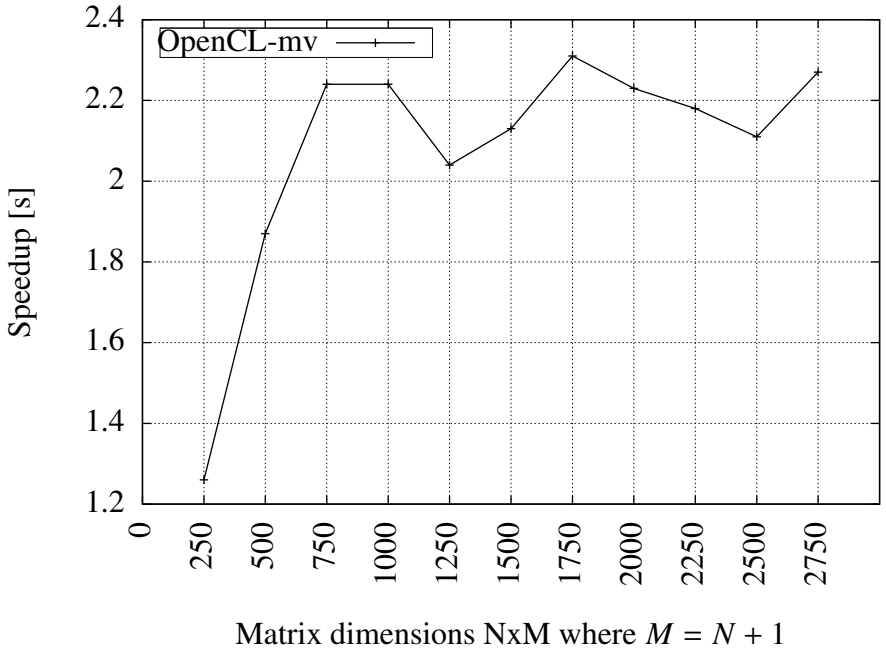


Figure D.4: SAOi speedups for OpenCL-mv are compared to Netlib BLAS as baseline.

Table D.1: Speedup table for NetLib BLAS vs. OpenCL-mv routines. CPU gives the timing information.

			NetLib BLAS			OpenCL-mv			
n	m	k^*	f^*	h^*	CPU	f^*	h^*	CPU	Speedup
250	250	11	54067.871	2.20E-06	5.49	54067.872	2.61E-06	6.89	1.26
500	500	11	53891.632	8.18E-07	16.42	53891.632	1.26E-05	30.74	1.87
750	750	11	53832.593	2.57E-06	35.57	53832.593	3.58E-06	79.69	2.24
1000	1000	12	53803.008	5.57E-06	68.44	53803.008	1.60E-05	153.07	2.24
1250	1250	12	53785.233	9.89E-06	119.94	53785.233	9.43E-06	244.86	2.04
1500	1500	12	53773.373	4.53E-06	175.87	53773.373	3.63E-06	374.12	2.13
1750	1750	13	53764.895	3.83E-06	232.25	53764.895	2.41E-05	537.36	2.31
2000	2000	13	53758.534	1.55E-05	327.64	53758.534	4.51E-06	730.94	2.23
2250	2250	13	53753.584	2.93E-06	442.02	53753.583	4.15E-06	963.06	2.18
2500	2500	13	53749.623	8.73E-06	534.51	53749.623	3.88E-06	1129.27	2.11
2750	2750	13	53746.381	3.15E-05	647.24	53746.381	2.94E-05	1468.92	2.27

The most noticeable aspect in the results is the meagre speed increase of approximately 2×. This unimpressive result is explained by the fact that only half of the possible matrix-vector multiplications have been replaced. This is mirrored in the profile data, showing a speed increase of only 50% over the NetLib BLAS, compared to profile data of CUBLAS of approximately 95%. The addition of an inverse OpenCL-mv routine will therefore greatly benefit the performance. Notwithstanding the slow memory transfer time, this OpenCL routine is still competitive with the corresponding CUBLAS matrix-vector routine when replaced into the SAO program.

Appendix E

Source code

This Appendix provides a listing of selected source code.

E.1 Single precision inclusion

E.1.1 Single and double precision driver for E04NQF

```
      subroutine drive_e04nqf1 (norg, xorg, np, nq, f_a, c_a, h_a,
     &                          iact, nact, xlam, x_h,
     &                          acurv, bcurv, ccurv, acurvgrad,
     &                          bcurvgrad, ccurvgrad, x_l, x_u,
     &                          f, c, h, gf, gc, gh, ksubiter,
     &                          xkkt, flam, cstage0, astring,xlam_h,
     &                          nsx,ictrl,lctrl,rctrl,message,yhi,
     &                          outerloop)
!
      implicit          none
      include           'ctrl.h'
      character*90      astring
      logical           cstage0
      integer           np, nq, norg, ksubiter,sdr
      integer           nact, iact(nact), i, j
      double precision  xorg(norg), flam
      double precision  x_h(norg), x_l(norg), x_u(norg), xlam(np)
      double precision  acurv, bcurv(np), ccurv(nq)
      double precision  acurvgrad(norg), bcurvgrad(np,norg)
      double precision  ccurvgrad(nq,norg)
      double precision  f, c(np), h(nq), gf(norg), gc(np,norg)
      double precision  f_a, c_a(np), h_a(nq)
      double precision  xkkt, gh(nq,norg),yhi

      integer           nsx,message,outerloop
      double precision  xlam_h(norg)
```

```
      real sp_xorg(norg), sp_flam
      real sp_x_h(norg), sp_x_l(norg), sp_x_u(norg), sp_xlam(np)
      real sp_acurv, sp_bcurv(np), sp_ccurv(nq)
      real sp_acurvgrad(norg), sp_bcurvgrad(np,norg)
      real sp_ccurvgrad(nq,norg)
      real sp_f, sp_c(np), sp_h(nq), sp_gf(norg), sp_gc(np,norg)
      real sp_f_a, sp_c_a(np), sp_h_a(nq)
      real sp_xkkt, sp_gh(nq,norg), sp_yhi

      integer           n_rel, a_rowrel((norg+1)*np)
      integer           a_colrel((norg+1)*np)
      double precision x_rel(norg+1), x_urel(norg+1), x_lrel(norg+1)
      double precision gfrel(norg+1), wrel(norg+1)
      double precision gcrel(np,norg+1), x_lirel(norg+1)
      double precision a_valrel((norg+1)*np), xirel(norg+1)
      double precision x_uirel(norg+1), zrel(norg+1)

      real rsp_xorg(norg+1), rsp_flam
      real rsp_x_h(norg), rsp_x_l(norg+1)
      real rsp_x_u(norg+1), rsp_xlam(np)
      real rsp_acurv, rsp_bcurv(np), rsp_ccurv(nq)
      real rsp_acurvgrad(norg+1), rsp_bcurvgrad(np,norg)
      real rsp_ccurvgrad(nq,norg), rsp_gc(np,norg+1)
      real rsp_f, rsp_c(np), rsp_h(nq), rsp_gf(norg+1)
      real rsp_f_a, rsp_c_a(np), rsp_h_a(nq)
      real rsp_xkkt, rsp_gh(nq,norg)

      include          'ctrl_get.inc'
! sd sets whether single or double precision e04nqf solver will be used
! [2 = double precision]     --     [1 = single precision]
! [3 = double precision relaxed] -- [4 = single precision relaxed]
!       sd = 1
      if (pen1.eq.0.d0) then
        sdr = 1
      else
        sdr = 2
      endif

      if (sd.eq.2) then ! do double presision

!  call the NAG e04nqf solver to solve the equivalent diagonal QP prob
        call drive_e04nqf_d (norg, xorg, np, nq, f_a, c_a, h_a,
     &                       iact, nact, xlam,
     &                       x_h, acurv, bcurv, ccurv,
     &                       acurvgrad, bcurvgrad, ccurvgrad,
     &                       x_l, x_u, f, c, h, gf, gc, gh, ksubiter,
     &                       xkkt, flam, cstage0, astring,xlam_h,
     &                       nsx,ictrl,lctrl,rctrl,message,yhi,
     &                       outerloop)

      elseif (sd.eq.1) then ! do single precision
!
! Make variables to be passed reals
```

```
        sp_xorg = real(xorg)
        sp_flam = real(flam)
        sp_x_h = real(x_h)
        sp_x_l = real(x_l)
        sp_x_u = real(x_u)
        sp_xlam = real(xlam)
        sp_acurv = real(acurv)
        sp_bcurv = real(bcurv)
        sp_ccurv = real(ccurv)
        sp_acurvgrad = real(acurvgrad)
        sp_bcurvgrad = real(bcurvgrad)
        sp_ccurvgrad = real(ccurvgrad)
        sp_f = real(f)
        sp_c = real(c)
        sp_h = real(h)
        sp_gf = real(gf)
        sp_gc = real(gc)
        sp_f_a = real(f_a)
        sp_c_a = real(c_a)
        sp_h_a = real(h_a)
        sp_xkkt = real(xkkt)
        sp_gh = real(gh)
        sp_yhi = real(yhi)

! call drive_sp_e04nqf1 and make double again
        call drive_e04nqf_s (norg, sp_xorg, np, nq, sp_f_a, sp_c_a,
     &                       sp_h_a, iact, nact, sp_xlam,sp_ x_h,
     &                       sp_acurv, sp_bcurv, sp_ccurv, sp_acurvgrad,
     &                       sp_bcurvgrad, sp_ccurvgrad, sp_x_l, sp_x_u,
     &                       sp_f, sp_c, sp_h, sp_gf, sp_gc, sp_gh,
     &                       ksubiter, sp_xkkt, sp_flam, cstage0,
     &                       astring,xlam_h,
     &                       nsx,ictrl,lctrl,rctrl,message,sp_yhi,
     &                       outerloop)

! Make returns double
        xorg = dble(sp_xorg)
        flam = dble(sp_flam)
        x_h = dble(sp_x_h)
        x_l = dble(sp_x_l)
        x_u = dble(sp_x_u)
        xlam = dble(sp_xlam)
        acurv = dble(sp_acurv)
        bcurv = dble(sp_bcurv)
        ccurv = dble(sp_ccurv)
        acurvgrad = dble(sp_acurvgrad)
        bcurvgrad = dble(sp_bcurvgrad)
        ccurvgrad = dble(sp_ccurvgrad)
        f = dble(sp_f)
        c = dble(sp_c)
        h = dble(sp_h)
        gf = dble(sp_gf)
        gc = dble(sp_gc)
        f_a = dble(sp_f_a)
```

```
136           c_a = dble(sp_c_a)
              h_a = dble(sp_h_a)
              xkkt = dble(sp_xkkt)
              gh = dble(sp_gh)
              yhi = dble(sp_yhi)
141
            elseif (sd.eq.3) then ! do double precision relaxed


              n_rel = norg+1
146           do i=1,norg
                x_rel(i) = xorg(i)
                x_urel(i) = x_u(i)
                x_lrel(i) = x_l(i)
                gfrel(i) = gf(i)
151           enddo
              x_rel(n_rel) = 0.d0
              x_urel(n_rel) = 1.d20
              x_lrel(n_rel) = 0.d0
              gfrel(n_rel) = pen1
156
              do i=1,np
                do j=1,norg
                  gcrel(i,j) = gc(i,j)
                enddo
161           enddo
              do j=1,np
                gcrel(j,n_rel) = -1.d0
              enddo

166           do i=1,norg
                wrel(i) = acurvgrad(i)
              enddo
              wrel(n_rel) = pen2

171   !         nsx =  n_rel*np

      !  call the NAG e04nqf solver to solve the equivalent diagonal QP prob
              call drive_e04nqf_d (n_rel, x_rel, np, nq, f_a, c_a, h_a,
           &                       iact, nact, xlam,
176        &                       x_h, acurv, bcurv, ccurv,
           &                       wrel, bcurvgrad, ccurvgrad,
           &                       x_lrel, x_urel, f, c, h, gfrel, gcrel,
           &                       gh, ksubiter,
           &                       xkkt, flam, cstage0, astring,xlam_h,
181        &                       nsx,ictrl,lctrl,rctrl,message,yhi,
           &                       outerloop)
              do i=1,norg
                xorg(i) = x_rel(i)
              enddo
186
            elseif (sd.eq.4) then ! do single precision relaxed

              n_rel = norg+1
```

```
191         do i=1,norg
              x_rel(i) = xorg(i)
              x_urel(i) = x_u(i)
              x_lrel(i) = x_l(i)
              gfrel(i) = gf(i)
196         enddo
            x_rel(n_rel) = 0.d0
            x_urel(n_rel) = 1.d20
            x_lrel(n_rel) = 0.d0
            gfrel(n_rel) = pen1
201
            do i=1,np
              do j=1,norg
                gcrel(i,j) = gc(i,j)
              enddo
206         enddo
            do j=1,np
              gcrel(j,n_rel) = -1.d0
            enddo

211         do i=1,norg
              wrel(i) = acurvgrad(i)
            enddo
            wrel(n_rel) = pen2

216 ! Make variables to be passed reals
            rsp_xorg = real(x_rel)
            rsp_flam = real(flam)
            rsp_x_h = real(x_h)
            rsp_x_l = real(x_lrel)
221         rsp_x_u = real(x_urel)
            rsp_xlam = real(xlam)
            rsp_acurv = real(acurv)
            rsp_bcurv = real(bcurv)
            rsp_ccurv = real(ccurv)
226         rsp_acurvgrad = real(wrel)
            rsp_bcurvgrad = real(bcurvgrad)
            rsp_ccurvgrad = real(ccurvgrad)
            rsp_f = real(f)
            rsp_c = real(c)
231         rsp_h = real(h)
            rsp_gf = real(gfrel)
            rsp_gc = real(gcrel)
            rsp_f_a = real(f_a)
            rsp_c_a = real(c_a)
236         rsp_h_a = real(h_a)
            rsp_xkkt = real(xkkt)
            rsp_gh = real(gh)

    ! call drive_sp_e04nqf1 and make double again
241         call drive_e04nqf_s (norg, rsp_xorg, np, nq, rsp_f_a, rsp_c_a,
        &                  rsp_h_a, iact, nact, rsp_xlam,rsp_ x_h,
        &                  rsp_acurv, rsp_bcurv, rsp_ccurv, rsp_acurvgrad,
```

```
     &                  rsp_bcurvgrad, rsp_ccurvgrad, rsp_x_l, rsp_x_u,
     &                  rsp_f, rsp_c, rsp_h, rsp_gf, rsp_gc, rsp_gh,
     &                  ksubiter, rsp_xkkt, rsp_flam, cstage0,
     &                  astring,xlam_h,
     &                  nsx,ictrl,lctrl,rctrl,message,yhi,
     &                  outerloop)

! Make returns double
        x_rel = dble(rsp_xorg)
        flam = dble(rsp_flam)
        x_h = dble(rsp_x_h)
        x_lrel = dble(rsp_x_l)
        x_urel = dble(rsp_x_u)
        xlam = dble(rsp_xlam)
        acurv = dble(rsp_acurv)
        bcurv = dble(rsp_bcurv)
        ccurv = dble(rsp_ccurv)
        wrel = dble(rsp_acurvgrad)
        bcurvgrad = dble(rsp_bcurvgrad)
        ccurvgrad = dble(rsp_ccurvgrad)
        f = dble(rsp_f)
        c = dble(rsp_c)
        h = dble(rsp_h)
        gfrel = dble(rsp_gf)
        gcrel = dble(rsp_gc)
        f_a = dble(rsp_f_a)
        c_a = dble(rsp_c_a)
        h_a = dble(rsp_h_a)
        xkkt = dble(rsp_xkkt)
        gh = dble(rsp_gh)

        do i=1,norg
          xorg(i) = x_rel(i)
        enddo
      else
        stop 'Precision not defined!'
      endif

      return
      end
```

E.1.2 Single and double precision driver for LSQP

```
      subroutine drive_galQP (n,x,m,nq,f_a,c_a,h_a,xlam,x_h,
     &                        acurvn,bcurvn,ccurvn,
     &                        x_l,x_u,f,c,h,gf,gc,gh,ksubiter,
     &                        xkkt,flam,cstage0,ostring,xlam_h,
     &                        nsx,ictrl,lctrl,rctrl,message,yhi,
     &                        outerloop,n2,n3,nm,kmn,xm,gfm,gcm)
      implicit none
```

```
      include           'ctrl.h'
      character*16      A_string,H_string
      character*90      ostring
      logical           kkt_local,cstage0
      integer           m,nq,n,n2,n3,ierr,a_ne,h_ne,ksubiter,outerloop
      integer           i,j,k,QPform_A,QPform_H,sdr,relaxed,nsx,message
!
      integer           A_row(n*m),A_col(n*m),A_ptr(m+1)
      integer           H_row(n2),H_col(n2),H_ptr(n+1),nm,kmn
!
      double precision  w(n),xlamj,bji,gh(nq,n)
      double precision  x(n),xi(n),x_li(n),x_ui(n)
      double precision  x_h(n),x_l(n),x_u(n),xlam(m)
      double precision  acurvn(n),bcurvn(m,n),gc_h(m,n),gf_h(n)
      double precision  ccurvn(nq,n),f,c(m),h(nq),gf(n),gc(m,n)
      double precision  f_a,c_a(m),h_a(nq),xkkt,gf_a(n),gc_a(m,n)
      double precision  y(m),z(n),xlam_h(m),yhi
      double precision  flam,c_l(m),c_u(m),A_val(n*m),g(n)
!
      double precision  H_val(n2)
!
      double precision  xm(n,nm),gfm(n,nm),gcm(m,n,nm)
!
      integer           nr,A_rowrel((n+1)*m),A_colrel((n+1)*m)
      integer           H_rowrel(n3),H_colrel(n3),H_ptrrel(n+2)
      double precision  xr(n+1),xr_u(n+1),xr_l(n+1),gfr(n+1),wrel(n+1)
      double precision  gcr(m,n+1),x_lirel(n+1),x_uirel(n+1)
      double precision  A_valrel((n+1)*m),xirel(n+1),zrel(n+1)
      double precision  H_valrel(n3)
!
      real              f_a_s, f_s, c_l_s(m), c_u_s(m), x_li_s(n)
      real              x_ui_s(n), xi_s(n), gf_s(n), w_s(n)
      real              xlam_s(m), z_s(n), a_val_s(n*m)
!
      real              H_val_s(n2)
      real              H_valrel_s(n3)
!
      real              gfr_s(n+1)
      real              wrel_s(n+1)
      real              x_lirel_s(n+1),x_uirel_s(n+1)
      real              a_valrel_s((n+1)*m),xirel_s(n+1),zrel_s(n+1)
!
      include           'ctrl_get.inc'
!
      ierr = 0
      QPform_A = 2 ! 0 = dense / 1 = coordinate / 2 = sparse_by_rows
      QPform_H = 3 ! 0 = dense / 1 = coordinate / 2 = sparse_by_rows
!                            / 3 = diagonal
!
      if (subsolver.eq.25) QPform_H = 3
!
      if (pen1.eq.0.d0) then
        sdr = 1
      else
```

```
62          sdr = 2
         endif

         if (sdr.eq.1) then
!
67          call QP_pre (n,m,a_ne,c,acurvn,
     &                 bcurvn,xlam,gc,x,xi,x_l,x_u,x_li,x_ui,
     &                 z,c_l,c_u,w,A_row,A_col,A_val,A_ptr,
     &                 A_string,QPform_A,n,QPform_H,H_string,
     &                 H_row,H_col,H_ptr,H_val,h_ne,ictrl,
72   &                 lctrl,rctrl,nm,kmn,xm,gfm,gcm,outerloop)

          if     (sd.eq.1) then
!           call the Galahad QP solvers (single)
            f_a_s = real(f_a)
77          f_s = real(f)
            c_l_s = real(c_l)
            c_u_s = real(c_u)
            x_li_s = real(x_li)
            x_ui_s = real(x_ui)
82          xi_s = real(xi)
            gf_s = real(gf)
            w_s = real(w)
            xlam_s = real(xlam)
            z_s = real(z)
87          a_val_s = real(a_val)
            H_val_s = real(H_val)
!

            if (subsolver.eq.25) then
92            call galahad_lsqp_s (n,m,a_ne,f_a_s,c_l_s,c_u_s,x_li_s,
     &                    x_ui_s,a_row,a_col,a_ptr,xi_s,gf_s,
     &                    w_s,xlam_s,z_s,a_val_s,A_string,ierr,
     &                    ksubiter,f_s)

97          elseif (subsolver.eq.26) then
              call galahad_qpa_s (n,m,a_ne,f_a_s,c_l_s,c_u_s,x_li_s,
     &                          x_ui_s,A_row,A_col,A_ptr,xi_s,gf_s,
     &                          w_s,xlam_s,z_s,A_val_s,A_string,ierr,
     &                          ksubiter,H_string,H_row,H_col,
102  &                          H_ptr,H_val_s,h_ne,f_s)

            elseif (subsolver.eq.27) then
              call galahad_qpb_s (n,m,a_ne,f_a_s,c_l_s,c_u_s,x_li_s,
     &                          x_ui_s,A_row,A_col,A_ptr,xi_s,gf_s,
107  &                          w_s,xlam_s,z_s,A_val_s,A_string,ierr,
     &                          ksubiter,H_string,H_row,H_col,
     &                          H_ptr,H_val_s,h_ne,f_s)

            elseif (subsolver.eq.28) then
112           call galahad_qpc_s (n,m,a_ne,f_a_s,c_l_s,c_u_s,x_li_s,
     &                          x_ui_s,A_row,A_col,A_ptr,xi_s,gf_s,
     &                          w_s,xlam_s,z_s,A_val_s,A_string,ierr,
     &                          ksubiter,H_string,H_row,H_col,
```

```
     &                          H_ptr,H_val_s,h_ne,f_s)
117
          endif
!
          f_a = dble(f_a_s)
          f = dble(f_s)
122       xi = dble(xi_s)
          xlam = dble(xlam_s)
!
        elseif (sd.eq.2) then
! call the Galahad QP solvers (double)
127       if     (subsolver.eq.25) then
            call galahad_lsqp_d (n,m,a_ne,f_a,c_l,c_u,x_li,
     &                           x_ui,A_row,A_col,A_ptr,xi,gf,
     &                           w,xlam,z,A_val,A_string,ierr,
     &                           ksubiter,f)
132
          elseif (subsolver.eq.26) then
            call galahad_qpa_d  (n,m,a_ne,f_a,c_l,c_u,x_li,
     &                           x_ui,A_row,A_col,A_ptr,xi,gf,
     &                           w,xlam,z,A_val,A_string,ierr,
137  &                           ksubiter,H_string,H_row,H_col,
     &                           H_ptr,H_val,h_ne,f)

          elseif (subsolver.eq.27) then
            call galahad_qpb_d  (n,m,a_ne,f_a,c_l,c_u,x_li,
142  &                           x_ui,A_row,A_col,A_ptr,xi,gf,
     &                           w,xlam,z,A_val,A_string,ierr,
     &                           ksubiter,H_string,H_row,H_col,
     &                           H_ptr,H_val,h_ne,f)

147       elseif (subsolver.eq.28) then
            call galahad_qpc_d  (n,m,a_ne,f_a,c_l,c_u,x_li,
     &                           x_ui,A_row,A_col,A_ptr,xi,gf,
     &                           w,xlam,z,A_val,A_string,ierr,
     &                           ksubiter,H_string,H_row,H_col,
152  &                           H_ptr,H_val,h_ne,f)

          endif
        endif
!
157   elseif (sdr.eq.2) then
!
        nr = n+1
        do i=1,n
          xr(i) = x(i)
162       xr_u(i) = x_u(i)
          xr_l(i) = x_l(i)
          gfr(i) = gf(i)
        enddo
        xr(nr) = 0.d0
167     xr_u(nr) = ymax
        xr_l(nr) = 0.d0
        gfr(nr) = pen1
```

```
!
        do j=1,m
172       do i=1,n
            gcr(j,i) = gc(j,i)
          enddo
        enddo
        do j=1,m
177       gcr(j,nr) = -1.d0
        enddo
!
        call QP_pre (nr,m,a_ne,c,acurvn,bcurvn,xlam,gcr,xr,xirel,xr_l,
     &               xr_u,x_lirel,x_uirel,zrel,c_l,c_u,wrel,A_rowrel,
182  &               A_colrel,A_valrel,A_ptr,A_string,QPform_A,n,
     &               QPform_H,H_string,H_rowrel,H_colrel,H_ptrrel,
     &               H_valrel,h_ne,ictrl,lctrl,rctrl,nm,kmn,xm,gfm,gcm,
     &               outerloop)
!
187     if     (sd.eq.1) then
!       call the Galahad QP solvers (single)
          f_a_s = real(f_a)
          f_s = real(f)
          c_l_s = real(c_l)
192       c_u_s = real(c_u)
          x_lirel_s = real(x_lirel)
          x_uirel_s = real(x_uirel)
          xirel_s = real(xirel)
          gfr_s = real(gfr)
197       wrel_s = real(wrel)
          xlam_s = real(xlam)
          zrel_s = real(zrel)
          a_valrel_s = real(a_valrel)
          H_valrel_s = real(H_valrel)
202
          if (subsolver.eq.25) then
            call galahad_lsqp_s (nr,m,a_ne,f_a_s,c_l_s,c_u_s,x_lirel_s,
     &                 x_uirel_s,A_rowrel,A_colrel,A_ptr,xirel_s,gfr_s,
     &                 wrel_s,xlam_s,zrel_s,a_valrel_s,A_string,ierr,
207  &                 ksubiter,f_s)

          elseif (subsolver.eq.26) then
            call galahad_qpa_s (nr,m,a_ne,f_a_s,c_l_s,c_u_s,x_lirel_s,
     &                          x_uirel_s,A_rowrel,A_colrel,A_ptr,
212  &                          xirel_s,gfr_s,wrel_s,xlam_s,zrel_s,
     &                          A_valrel_s,A_string,ierr,ksubiter,
     &                          H_string,H_rowrel,H_colrel,H_ptr,
     &                          H_valrel_s,h_ne,f_s)

217       elseif (subsolver.eq.27) then
            call galahad_qpb_s (nr,m,a_ne,f_a_s,c_l_s,c_u_s,x_lirel_s,
     &                          x_uirel_s,A_rowrel,A_colrel,A_ptr,
     &                          xirel_s,gfr_s,wrel_s,xlam_s,zrel_s,
     &                          A_valrel_s,A_string,ierr,ksubiter,
222  &                          H_string,H_rowrel,H_colrel,H_ptr,
     &                          H_valrel_s,h_ne,f_s)
```

```
          elseif (subsolver.eq.28) then
            call galahad_qpc_s (nr,m,a_ne,f_a_s,c_l_s,c_u_s,x_lirel_s,
     &                          x_uirel_s,A_rowrel,A_colrel,A_ptr,
     &                          xirel_s,gfr_s,wrel_s,xlam_s,zrel_s,
     &                          A_valrel_s,A_string,ierr,ksubiter,
     &                          H_string,H_rowrel,H_colrel,H_ptr,
     &                          H_valrel_s,h_ne,f_s)

          endif

          f_a = dble(f_a_s)
          f = dble(f_s)
          xi = dble(xirel_s)
          xlam = dble(xlam_s)

!
        elseif (sd.eq.2) then

! call the Galahad QP solvers (double)
          if     (subsolver.eq.25) then
            call galahad_lsqp_d (nr,m,a_ne,f_a,c_l,c_u,x_lirel,
     &                  x_uirel,A_rowrel,A_colrel,A_ptr,xirel,
     &                  gfr,wrel,xlam,zrel,A_valrel,A_string,ierr,
     &                  ksubiter,f)

          elseif (subsolver.eq.26) then
            call galahad_qpa_d  (nr,m,a_ne,f_a,c_l,c_u,x_lirel,
     &                  x_uirel,A_rowrel,A_colrel,A_ptr,xirel,
     &                  gfr,wrel,xlam,zrel,A_valrel,A_string,ierr,
     &                  ksubiter,H_string,H_rowrel,H_colrel,
     &                  H_ptrrel,H_valrel,h_ne,f)

          elseif (subsolver.eq.27) then
            call galahad_qpb_d  (nr,m,a_ne,f_a,c_l,c_u,x_lirel,
     &                  x_uirel,A_rowrel,A_colrel,A_ptr,xirel,
     &                  gfr,wrel,xlam,zrel,A_valrel,A_string,ierr,
     &                  ksubiter,H_string,H_rowrel,H_colrel,
     &                  H_ptrrel,H_valrel,h_ne,f)

          elseif (subsolver.eq.28) then
            call galahad_qpc_d  (nr,m,a_ne,f_a,c_l,c_u,x_lirel,
     &                  x_uirel,A_rowrel,A_colrel,A_ptr,xirel,
     &                  gfr,wrel,xlam,zrel,A_valrel,A_string,ierr,
     &                  ksubiter,H_string,H_rowrel,H_colrel,
     &                  H_ptrrel,H_valrel,h_ne,f)

          endif
        endif
!
        do i=1,n
         xi(i) = xirel(i)
        enddo
        yhi = xirel(nr)
```

```
!
      else
!
        stop ' sdr invalid in drive_gqp'
!
      endif

! update multipliers
      do j=1,m
        xlam(j)= min(dabs(xlam(j)),biglam)
      enddo

! update x in current point
      do i=1,n
        x(i)=min(x_u(i),max(x_l(i),x(i)+xi(i)))
      enddo
!
      message = ierr

      if (isNaN(f_a)) then
        write(*,*) ' '
        write(*,*) ' STOP: the subproblem returned fapprox = ',f_a
        write(*,*) ' '
      endif

! overwrite f_a here if not used
      call fun_a (n,x,f_a,x_h,-1.d0,acurvn,f,gf)

      return
      end subroutine drive_galQP
```

E.2 Parallel Dual and QP code

Three subsolvers in parallel; two are based on quadratic approximations `LSQP` and `E04NQF`, one is based on a quadratic dual approximation `l-BFGS-b`. Algorithm 3 displays the algorithm that has been applied to solve a test problem with all three solvers simultaneously.

```
! SAOi:
! SAOi: Sat May 02 09:04:51 SAST 2009, Albert Groenwold, Stellenbosch
! SAOi: The secondary SAO driver
! SAOi:


      subroutine sao_dense(n,ni,ne,x,x_l0,x_u0,timef,timeg,times,
     &                     f,viol,xkkt,kloop,llooptot,
     &                     ictrl,lctrl,rctrl,cctrl,warn,nfe,nge,
     &                     feasible,iuser,luser,cuser,ruser,
     &                     nnz,eqn,lin,nw,nm,u,ce,cd,cm,i97,j97,
     &                     raninit,mxloop,ierr)
      implicit          none
```

```
      include           'ctrl.h'
      logical           eqn(*), lin(*),raninit
      integer           i97,j97,ivec,lenr
      double precision  u(97),ce,cd,cm
      character*90      ostring
      integer           kloopmax,lloopmax,kloop,llooptot,lloop,ig,ns,nnz
      integer           iactblo,iactbhi,iactc,ipr,n5,i,j,nact,nfe,nge
      integer           n,ni,ne,iact(ni),ksubitertot,infilter,ksubiter
      integer           outerloop,message,ifree,nw,mxloop,ierr
      integer           Acol(1),ncol(1),n1,nm,kmn
      double precision  x(n),x0(n),xlam(ni+ne),c_h(ni),h_h(ne),v_h
      double precision  h_a(ne),c_a(ni),x_h(n),x_h2(n),f_h
      double precision  gf_h(n),gc_h(nnz),gh_h(ne,n),x_l0(n),x_u0(n)
      double precision  acurv,bcurv(ni),ccurv(ne),x_l(n),x_u(n)
      double precision  acurvn(n),bcurvn(ni,n),ccurvn(ne,n)
      double precision  f,c(ni),h(ne),gf(n),filter_list(2,ictrl(05)+1)
      double precision  gc(nnz),gh(ne,n),yhi,cplus_norm_h
      double precision  timef1,timeg1,times1,timef2,timeg2,times2
      double precision  eps_locala,eps_localb,rfd,dml_inf0, rho
      double precision  current_tol,current_kkt,f_a,cplus_norm,norm2b
      double precision  SAOi_seconds,epsmch,dpmeps,viol,drange
      double precision  pred_f,real_f,qindicator,delxnormc,xkkt,flam
      double precision  xlamsml,xlambig,phibw,timef,timeg,times,xkktl
      double precision  Li(n),Ui(n),Li_h(n),Ui_h(n),delxnormi,norm2f
      double precision  hsum,xlam_h(ni+ne),s(n),xlamw(nw),xlamw_h(nw)
      double precision  xm(n,nm),gfm(n,nm),gcm(ni,n,nm)
      logical           warn,finalstg,accept,cstage0,init,do_diff
      logical           feasible
      save              timef1,timeg1,times1,timef2,timeg2,times2

      integer           flag,tid
      integer           omp_get_num_threads, omp_get_thread_num
      double precision  wtime0,wtime1,wtime2,wtime3,omp_get_wtime
      double precision  wtimef,wtimei,timediff

      integer           msg_dual, msg_e04, msg_gal
      double precision  x_dual(n),x_e04(n),x_gal(n),xlam_dual(ni+ne)
      double precision  xlam_e04(ni+ne),xkktl_dual,xkktl_e04, yhi_e04
      double precision  xlam_gal(ni+ne),xkktl_gal,yhi_gal,flam_gal
      double precision  flam_dual,flam_e04,f_a_e04,f_a_dual, yhi_dual
      double precision  f_a_gal
      character*7       solvestring

      include           'ctrl_get.inc'

!  initialize counters and a few other thingies
      tot_time_lost    = 0.d0
      ierr             = 0
      n1               = n+1
      message          = -200
      init             = .true.
      yhi              = 0.d0
      kloopmax         = outermax
      lloopmax         = innermax
```

```
      kloop           = 0
      lloop           = 0
      llooptot        = 0
      outerloop       = 0
      eps_locala      = 1.d-7
      eps_localb      = 0.d+0
      ksubitertot     = 0
      rfd             = big
      n5              = 5
      cstage0         = .false.
      finalstg        = .false.
      dml_inf0        = dml_infinity
      rho             = dml_infinity
      do i=1,n
        x0(i)         = x(i)
        x_l(i)        = x_l0(i)
        x_u(i)        = x_u0(i)
      enddo
      current_tol     = 1.d0
      current_kkt     = 1.d0
      epsmch          = dpmeps()
      acurv           = 0.d0
      do j=1,ni
        bcurv(j)      = 0.d0
      enddo
      do j=1,ne
        ccurv(j)      = 0.d0
      enddo
      do i=1,n
        acurvn(i)     = 0.d0
      enddo
      do i=1,n
        do j=1,ni
          bcurvn(j,i) = 0.d0
        enddo
      enddo
      do i=1,n
        do j=1,ne
          ccurvn(j,i) = 0.d0
        enddo
      enddo

!  write a few headers
      if (iprint.ge.1) write(6,6011)
      if (iprint.ge.3) write(8,5000)
      write(9,6011)
      write(10,7500)
      write(11,7500)
      if (debug) write(12,6001)
      if (iprint.ge.3) write(13,6002)
      if (check_grad) open (17,file='CheckSAOi_grad.out')
!
      if (ne.ne.0) goto 4000
!
```

```
123 !  determine function and constraint values at starting point
       timef1 = SAOi_seconds()
       call SAOi_funcs (n,ni,ne,x,f,c,h,iuser,luser,cuser,ruser,
      &                 eqn,lin,ictrl,lctrl,rctrl,cctrl)
       nfe=nfe+1
128    timef2 = SAOi_seconds()
       timef=timef+(timef2-timef1)

! catch floating point exceptions
       if (isNaN(f)) then ! .or.dabs(f).gt.1.d20) then
133      write(*,*) ' Terminal: the objective function returned ',f
         ierr=-500
         return
       endif
       do j=1,ni
138      if (isNaN(c(j))) then ! .or.dabs(c(j)).gt.1.d20) then
           write(*,*) ' Terminal: constraint function',j,
      &               ' returned ',c(j)
           ierr=-501
           return
143      endif
       enddo

!  determine gradients of function and constraints at starting point
       timeg1 = SAOi_seconds()
148    call formgrad (n,x,ni,ne,f,c,h,gf,gc,gh,ictrl,lctrl,rctrl,cctrl,
      &               nfe,nge,do_diff,iuser,luser,cuser,ruser,nnz,
      &               Acol,ncol,eqn,lin)
       timeg2 = SAOi_seconds()
       timeg=timeg+(timeg2-timeg1)
153
!  calculate the active set on the sao level
       call form_act(ni,c,iact,nact,ictrl,lctrl,rctrl)

!  initialize the dual variables for the Falk dual
158    do i=1,ni
         xlam(i)=0.d0
         xlam_h(i)=0.d0
       enddo

163 !  initialize the dual variables for the Wolfe dual
       do i=1,nw
         xlamw(i)=0.d0
         xlamw_h(i)=0.d0
       enddo
168
!  determine if starting point is feasible
       if (unconstrained) then
         call SAOistatus(n,ni,iactblo,iactbhi,iactc,cplus_norm,xlam,
      &                  viol,c,x,x_l0,x_u0,5,1,hsum,ictrl,lctrl,rctrl,
173   &                  feasible)
       else
         call SAOistatus(n,ni,iactblo,iactbhi,iactc,cplus_norm,xlam,
```

```
     &                  viol,c,x,x_l0,x_u0,1,1,hsum,ictrl,lctrl,rctrl,
     &                  feasible)
178    endif

!  adjust the relaxation variable upper bound
       ymax = max(0.d0,viol)
       rctrl(25) = ymax
183
!  output some subproblem stuff
       ostring = ' the starting point '
       if (debug) write(12,8500) 0,0,0,0.d0,0.d0,f,viol,ostring

188 !  conditionally initialize cstage0
       if (viol.gt.feaslim) cstage0=.true.

!  construct approximate diagonal hessian
       call diaHess(n,x,ni,ne,x_h,x_h2,acurv,bcurv,ccurv,acurvn,
193  &             bcurvn,ccurvn,x_l,x_u,f,c,h,gf,gc,gh,
     &             gf_h,gc_h,gh_h,Li,Ui,Li_h,Ui_h,f_h,c_h,h_h,s,
     &             ictrl,lctrl,rctrl,cctrl,outerloop,iuser,luser,
     &             cuser,ruser)

198 !  construct the lbfgs vectors
         call strv(n,ni,nm,outerloop,kmn,x,xm,gf,gfm,gc,gcm)

!  store first iterate
       ig=0
203    if (approx_c.eq.2.or.approx_c.eq.3.or.approx_c.eq.5) ig=1
       call store_iter (n,ni,ne,f_h,f,v_h,viol,x_h,x_h2,x,c_h,c,
     &                  cplus_norm,cplus_norm_h,gf_h,gf,gc_h,gc,
     &                  h_h,h,gh_h,gh,xlam,xlam_h,nw,xlamw,xlamw_h,
     &                  1,ig)
208
!  write some output
       if (iprint.ge.1) write (*,7010) kloop,lloop,f,viol,feasible
       write (9,7000) kloop,lloop,f,viol,feasible

213 !  possibly do specialized things
       call SAOi_Special (n,ni,ne,x,f,c,h,iuser,luser,cuser,ruser,
     &                    eqn,lin,ictrl,lctrl,rctrl,cctrl,x_l0,x_u0,
     &                    kloop,.false.,xkkt,viol,iactblo,iactbhi,
     &                    iactc)
218
!  prepare for the outer optimization loops
       kloop=1

!  flush a buffer the fortran way...
223    close(9)
       open (9 ,file='History.out',status='old',position='append')
       if (debug) then
         close(12)
         open (12 ,file='Subproblems.out',status='old',position='append')
228    endif
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!                                                                             !
!                start the outer optimization loop                            !
!                                                                             !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

      do while (kloop.le.kloopmax)
!
        call ranmar(rctrl(34),1,u,ce,cd,cm,i97,j97,raninit)
        outerloop=kloop
        lloop=0
!
        rho=dml_inf0
!
 110    continue
        do i=1,n
          drange=min(rangemax,rho*(x_u0(i)-x_l0(i)))
          x_l(i)=max(x(i)-drange,x_l0(i))
          x_u(i)=min(x(i)+drange,x_u0(i))
        enddo

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!                                                                             !
!              possibly start an inner conservative loop                      !
!                                                                             !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!  determine the sparsity of the Jacobian
        call sparser (n,ni,gf,gc,ns,ictrl,lctrl,rctrl,warn)

!  initialize and check for unconstrained problems
        ksubiter=0
        times1 = SAOi_seconds()
!
        if (unconstrained) then

          if (outerloop.eq.1) message = -50
          call uncstr (n, x, f_a, x_h, acurv, acurvn, x_l, x_u, f, gf,
     &                 ksubiter)

        else
! Reset flag to 0
          message = 0
          flag = 0
! Set the number of threads to be spawned
          call omp_set_num_threads(3)
          wtime0 = SAOi_seconds()

!$omp parallel default(shared)
!$omp+private(tid)
          tid = omp_get_thread_num()

!
!$omp sections
```

```
!$omp section
              f_a_dual = f_a
              x_dual = x
              xlam_dual = xlam
              xkktl_dual = xkktl
              flam_dual = flam
              msg_dual = message
              yhi_dual = yhi
              call drive_lbfgsbf (n,x_dual,ni,ne,f_a_dual,c_a,h_a,iact,
     &                            nact,xlam_dual,x_h,acurv,bcurv,ccurv,
     &                            acurvn,bcurvn,ccurvn,
     &                            x_l,x_u,f,c,h,gf,gc,gh,ksubiter,
     &                            xkktl_dual,flam_dual,cstage0,ostring,
     &                            xlam_h,ns,ictrl,lctrl,rctrl,msg_dual,
     &                            yhi_dual,outerloop,flag)
              wtime1 = SAOi_seconds()
              if (flag.le.0) then
                  write(*,*)  msg_dual, 'Dual'
!                 if (xkktl_dual.lt.kkt_tol) then
                if (xkktl_dual.lt.kkt_tol.or.msg_dual.eq.0) then
                  flag = 1
                  wtimei = SAOi_seconds()
                endif
              endif

!$omp flush(flag,f_a_dual,x_dual,xlam_dual,xkktl_dual,flam_dual,
!$omp+msg_dual)
!$omp section
              f_a_e04 = f_a
              x_e04 = x
              xlam_e04 = xlam
              xkktl_e04 = xkktl
              flam_e04 = flam
              msg_e04 = message
              yhi_e04 = yhi
!   call the NAG e04nqf solver to solve the equivalent diagonal QP prob
              call drive_e04nqf1 (n,x_e04,ni,ne,f_a_e04,c_a,h_a,iact,nact,
     &                            xlam_e04,x_h,acurv,bcurv,ccurv,
     &                            acurvn,bcurvn,ccurvn,
     &                            x_l,x_u,f,c,h,gf,gc,gh,ksubiter,
     &                            xkktl_e04,flam_e04,cstage0,ostring,
     &                            xlam_h,ns,ictrl,lctrl,rctrl,msg_e04,
     &                            yhi_e04,outerloop,flag)
              wtime2 = SAOi_seconds()
              if (flag.le.0) then
                  write(*,*)  msg_e04,'E04NQF'
!                 if (xkktl_e04.lt.kkt_tol) then
                if (xkktl_e04.lt.kkt_tol.or.msg_e04.eq.0
     &                                  .or.msg_e04.eq.5) then
                  flag = 2
                  wtimei = SAOi_seconds()
                endif
              endif
!$omp flush(flag,f_a_e04,x_e04,xlam_e04,xkktl_e04,flam_e04,msg_e04)
```

```
!$omp section
            f_a_gal = f_a
            x_gal = x
            xlam_gal = xlam
            xkktl_gal = xkktl
            flam_gal = flam
            msg_gal = message
            yhi_gal = yhi
!  call the Galahad QP solvers to solve the equivalent diagonal QP prob
            call drive_galQP (n,x_gal,ni,ne,f_a_gal,c_a,h_a,xlam_gal,
     &                        x_h,acurvn,bcurvn,ccurvn,x_l,x_u,f,
     &                        c,h,gf,gc,gh,ksubiter,
     &                        xkktl_gal,flam_gal,cstage0,ostring,xlam_h,
     &                        ns,ictrl,lctrl,rctrl,msg_gal,yhi_gal,
     &                        outerloop,(n*n+n)/2,(n1*n1+n1)/2,
     &                        nm,kmn,xm,gfm,gcm,ierr,flag,iact,nact)
            wtime3 = SAOi_seconds()
            if (flag.le.0) then
                write(*,*)  msg_gal, 'GALAHAD'
!                 if (xkktl_gal.lt.kkt_tol) then
              if (xkktl_gal.lt.kkt_tol.or.msg_gal.ge.0.or.msg_gal.eq.-5
     &            .or.msg_gal.eq.0) then
                flag = 3
                wtimei = SAOi_seconds()
              endif
            endif
!$omp flush(flag,f_a_gal,x_gal,xlam_gal,xkktl_gal,flam_gal,msg_gal)
!$omp end sections

!$omp end parallel

          wtimef = SAOi_seconds()

          time_lost = (wtimef - wtimei)
          tot_time_lost =  tot_time_lost + time_lost
          lost = tot_time_lost
!         Write a file for the tables in latex format
!           close(30)
!           open (30 ,file='mytimer.out',status='old',position='append')
!           write(30,'(2f10.2)') time_lost, tot_time_lost, lost

          if (flag.eq.1) then
!            write(*,*)   'Dual'
           solvestring = 'Dual'
           f_a   = f_a_dual
           x     = x_dual
           xlam  = xlam_dual
           xkktl = xkktl_dual
           flam  = flam_dual
           message = msg_dual
           yhi   = yhi_dual
          elseif (flag.eq.2) then
!            write(*,*)   'E04NQF'
           solvestring = 'E04NQF'
```

```
            f_a    = f_a_e04
            x      = x_e04
            xlam   = xlam_e04
            xkktl  = xkktl_e04
            flam   = flam_e04
            message = msg_e04
            yhi    = yhi_e04
          elseif (flag.eq.3) then
!           write(*,*)   'GALAHAD'
            solvestring = 'GALAHAD'
            f_a    = f_a_gal
            x      = x_gal
            xlam   = xlam_gal
            xkktl  = xkktl_gal
            flam   = flam_gal
            message = msg_gal
            yhi    = yhi_gal
          endif


        endif

! catch floating point exceptions
        if (isNaN(f_a)) then ! .or.dabs(f_a).gt.1.d20) then
          write(*,*) ' Terminal: the subproblem returned ',f_a
          ierr=-502
          return
        endif
        do j=1,ni
          if (isNaN(c_a(j))) then ! .or.dabs(c_a(j)).gt.1.d20) then
            write(*,*) ' Terminal: subconstraint function',j,
     &                 ' returned ',c_a(j)
            ierr=-503
            return
          endif
        enddo

!  return if there are errors
      if(ierr.lt.0) return

!  do some timing
        times2 = SAOi_seconds()
        times=times+(times2-times1)
        ksubitertot=ksubitertot+ksubiter

!  determine true function and constraint values
        timef1 = SAOi_seconds()
        call SAOi_funcs (n,ni,ne,x,f,c,h,iuser,luser,cuser,ruser,eqn,
     &                   lin,ictrl,lctrl,rctrl,cctrl)
        nfe=nfe+1
        timef2 = SAOi_seconds()
        timef=timef+(timef2-timef1)

! catch floating point exceptions
```

```
        if (isNaN(f)) then ! .or.dabs(f).gt.1.d20) then
          write(*,*) ' Terminal: the problem itself returned ',f
          ierr=-500
          return
        endif
        do j=1,ni
          if (isNaN(c(j))) then ! .or.dabs(c(j)).gt.1.d20) then
            write(*,*) ' Terminal: constraint function',j,
     &                 ' returned ',c(j)
            ierr=-501
            return
          endif
        enddo

!  determine if feasible
        if (unconstrained) then
          call SAOistatus(n,ni,iactblo,iactbhi,iactc,cplus_norm,xlam,
     &                    viol,c,x,x_l0,x_u0,5,-1,hsum,ictrl,lctrl,rctrl,
     &                    feasible)
        else
          call SAOistatus(n,ni,iactblo,iactbhi,iactc,cplus_norm,xlam,
     &                    viol,c,x,x_l0,x_u0,1,-1,hsum,ictrl,lctrl,rctrl,
     &                    feasible)
        endif
        accept=.true.

!  adjust the relaxation variable upper bound
      ymax = max(0.d0,viol)
      rctrl(25) = ymax

!  conditionally kill cstage0
        if (viol.lt.feaslim) cstage0=.false.

!  output some subproblem stuff
        if (debug)
     &    write(12,8500) kloop,lloop,ksubiter,flam,xkktl,f,viol,ostring

!   escape if relative step size is too small
        delxnormc = norm2b(n,x_h,x)
        delxnormi = norm2f(n,x_h,x)
        if (delxnormc.lt.xtol.or.delxnormi.lt.xtol_inf) goto 1234

!  escape if a feasible descent step was made
        if (conservative.and.f.lt.f_h.and.unconstrained) then
          goto 1234                     ! do not enforce conservatism
        endif

!  escape if a feasible descent step was made
        if (conservative.and.f.lt.f_h.and.feasible.and.allow_f) then
          goto 1234                     ! do not enforce conservatism
        endif

!  escape if an infeasible restoration step was made
        if (conservative.and.viol.lt.v_h.and.cstage0.and.allow_c) then
```

```
          goto 1234                        ! do not enforce conservatism
        endif

!  determine if a conservative inner loop is required
        if (conservative.and.kloop.gt.-1) then
          call conserve (n,ni,accept,f,f_a,c,c_a,eps_locala,
     &                   eps_localb,acurv,acurvn,bcurv,
     &                   bcurvn,rho,ictrl,lctrl,rctrl,lloop)
        endif

!  determine if a trust region step is required
        if (trust_region) then
          call filter_ise (n,ni,accept,f,f_h,f_a,viol,v_h,
     &                     c,c_h,c_a,kloop,acurv,
     &                     acurvn,bcurv,bcurvn,
     &                     eps_locala,eps_localb,
     &                     filter_list,infilter,init,rho,
     &                     ictrl,lctrl,rctrl,lloop)
          if (rho.eq.rho_l_min) then
            goto 1234 ! no further reduction possible
          endif
        endif

!  increment inner loop counters
        if (.not.accept.and.lloop.lt.lloopmax)  then
          lloop=lloop+1
          llooptot=llooptot+1

!  restore iterate
          call store_iter (n,ni,ne,f_h,f,v_h,viol,x_h,x_h2,x,c_h,c,
     &                     cplus_norm,cplus_norm_h,gf_h,gf,gc_h,gc,
     &                     h_h,h,gh_h,gh,xlam,xlam_h,nw,xlamw,xlamw_h,
     &                     -1,0)
          goto 110
        endif
!
 1234   continue ! escape from the inner loop

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!                                                                             !
!                       end the inner conservative loop                       !
!                                                                             !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!  determine if feasible
        if (unconstrained) then
          call SAOistatus(n,ni,iactblo,iactbhi,iactc,cplus_norm,xlam,
     &                    viol,c,x,x_l0,x_u0,5,1,hsum,ictrl,lctrl,rctrl,
     &                    feasible)
        else
          call SAOistatus(n,ni,iactblo,iactbhi,iactc,cplus_norm,xlam,
     &                    viol,c,x,x_l0,x_u0,1,1,hsum,ictrl,lctrl,rctrl,
     &                    feasible)
        endif
```

```
!  evaluate some filter thingies
        pred_f=f_h-f_a
        real_f=f_h-f
        if (pred_f.ne.0.d0) then
          qindicator=real_f/pred_f
        else
          qindicator=0.d0
        endif

!  calculate relative step sizes
        delxnormc = norm2b(n,x_h,x)
        delxnormi = norm2f(n,x_h,x)
        if (delxnormc.lt.xtol*1.d1.or.delxnormi.lt.xtol_inf*1.d1)
     &       finalstg=.true.

!  calculate relative function difference
        rfd=(f_h-f)/(1.d0+dabs(f_h))

!  determine gradients
        timeg1 = SAOi_seconds()
        call formgrad (n,x,ni,ne,f,c,h,gf,gc,gh,ictrl,lctrl,rctrl,cctrl,
     &                 nfe,nge,do_diff,iuser,luser,cuser,ruser,nnz,
     &                 Acol,ncol,eqn,lin)
        timeg2 = SAOi_seconds()
        timeg=timeg+(timeg2-timeg1)

!  construct the approximate diagonal hessian
        call diaHess(n,x,ni,ne,x_h,x_h2,acurv,bcurv,ccurv,acurvn,
     &               bcurvn,ccurvn,x_l,x_u,f,c,h,gf,gc,gh,
     &               gf_h,gc_h,gh_h,Li,Ui,Li_h,Ui_h,f_h,c_h,h_h,s,
     &               ictrl,lctrl,rctrl,cctrl,outerloop,iuser,luser,
     &               cuser,ruser)

!  construct the lbfgs vectors
        call strv(n,ni,nm,outerloop,kmn,x,xm,gf,gfm,gc,gcm)

!  store iterate
        ig=0
        if (approx_c.eq.2.or.approx_c.eq.3.or.approx_c.eq.5) ig=1
        call store_iter (n,ni,ne,f_h,f,v_h,viol,x_h,x_h2,x,c_h,c,
     &                   cplus_norm,cplus_norm_h,gf_h,gf,gc_h,gc,
     &                   h_h,h,gh_h,gh,xlam,xlam_h,nw,xlamw,xlamw_h,
     &                   1,ig)

!  calculate the active set on the subproblem level
        call form_act(ni,c,iact,nact,ictrl,lctrl,rctrl)

!  calculate the true KKT conditions
        call form_kkt (xkkt,n,ni,ne,x,iact,nact,x_l0,x_u0,
     &                 gf,gc,gh,xlam,xlamsml,xlambig,ifree,
     &                 ictrl,lctrl,rctrl)
!
        call fx_kkt (xkkt,viol,ifree,c,xlam,ni,ictrl,lctrl,rctrl)
```

```
608
      ! write some output
              if (iprint.ge.1) write (*,7002) kloop,lloop,f,viol,feasible,
           &  finalstg,
           &  rho,qindicator,delxnormc,rfd,
613        &  iactblo,iactbhi,iactc,message,solvestring

      ! write some more output
              write (9,7002) kloop,lloop,f,viol,feasible,
           &  finalstg,
618        &  rho,qindicator,delxnormc,rfd,
           &  iactblo,iactbhi,iactc,message,solvestring

      ! write even more output
              if (delxnormc.lt.current_tol) then
623             write(10,7600)current_tol,kloop,llooptot
                do i=1,15
                  if (current_tol/10.d0.lt.delxnormc) then
                    current_tol=current_tol/10.d0
                    exit
628               else
                    current_tol=current_tol/10.d0
                    write(10,7600)current_tol,kloop,llooptot
                  endif
                enddo
633           endif
              if (xkkt.lt.current_kkt) then
                write(11,7600)current_kkt,kloop,llooptot
                do i=1,15
                  if (current_kkt/10.d0.lt.xkkt) then
638                 current_kkt=current_kkt/10.d0
                    exit
                  else
                  current_kkt=current_kkt/10.d0
                  write(11,7600)current_kkt,kloop,llooptot
643               endif
                enddo
              endif

      !  exit do loop here on final iteration
648           if (kloop.eq.kloopmax
           &      .or. (delxnormc.le.xtol.and.current_kkt.le.kkt_min)
           &      .or. (delxnormi.le.xtol_inf.and.current_kkt.le.kkt_min)
           &      .or. current_kkt.le.kkt_tol) then
                if (iprint.ge.1) write(*,9600)
653             write(9,*) ' '
                if (kloop.eq.kloopmax) then
                  if (iprint.ge.1) write(*,6012)
                  write(9,6012)
                  call open_warn_file (warn)
658               write (14,6013)
                end if
                if (delxnormc.le.xtol.and.current_kkt.le.kkt_min) then
                  if (iprint.ge.1) write(*,6014)
```

```
            write(9,6014)
          end if
          if (current_kkt.le.kkt_tol) then
            if (iprint.ge.1) write(*,6015)
            write(9,6015)
          end if
          if (delxnormi.le.xtol_inf.and.current_kkt.le.kkt_min) then
            if (iprint.ge.1) write(*,6016)
            write(9,6016)
          end if
          if (iprint.ge.1) write(*,9600)
          write(9,*) ' '
          exit
        endif           ! stopping condition
!
        if (dabs(rfd).lt.ftol) then
          if (iprint.ge.1) write(*,9600)
          write(9,*) ' '
          if (iprint.ge.1) write(*,6017)
          write(9,6017)
          if (iprint.ge.1) write(*,9600)
          write(9,*) ' '
          exit
        endif
!

!  again flush some buffers the fortran way...
        close(9)
        open (9 ,file='History.out',status='old',position='append')
        if (debug) then
          close(12)
          open (12,file='Subproblems.out',status='old',
     &          position='append')
        endif

!  possibly do specialized things
        call SAOi_Special (n,ni,ne,x,f,c,h,iuser,luser,cuser,ruser,
     &                    eqn,lin,ictrl,lctrl,rctrl,cctrl,x_l0,x_u0,
     &                    kloop,.false.,xkkt,viol,iactblo,iactbhi,
     &                    iactc)

! increment outer loop counter
        kloop=kloop+1
!
      end do             ! while kloop.lt.kloopmax

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!                                                                       !
!                   end of outer optimization loop                      !
!                                                                       !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!  write a final summary
      do ipr=6,9,3
```

```
        if ((ipr.eq.6.and.iprint.ge.2).or.ipr.eq.9) then
          write(ipr,*) ' number of variables n    : ',n
          write(ipr,*) ' number of constraints ni : ',ni
          write(ipr,*) ' '
          write(ipr,*) ' Outer iterations k used  : ',kloop
          write(ipr,*) ' Inner iterations l used  : ',llooptot
          write(ipr,*) ' Subproblem evaluations   : ',ksubitertot
          write(ipr,*) ' '
          write(ipr,*) ' Machine precision        : ',epsmch
          write(ipr,*) ' '
          write(ipr,8101) f,viol
          write(ipr,8102) (x(i),i=1,min(n,n5))
          if (ni.gt.0) then
            write(ipr,8103) (c(i),i=1,min(ni,n5))
            write(ipr,8104) (xlam(i),i=1,min(ni,n5))
          endif
        endif
      enddo

!  calculate the true KKT conditions
      call form_kkt (xkkt,n,ni,ne,x,iact,nact,x_l0,x_u0,
     &               gf,gc,gh,xlam,xlamsml,xlambig,ifree,
     &               ictrl,lctrl,rctrl)
!
      call fx_kkt (xkkt,viol,ifree,c,xlam,ni,ictrl,lctrl,rctrl)

!  possibly do specialized things
        call SAOi_Special (n,ni,ne,x,f,c,h,iuser,luser,cuser,ruser,
     &                     eqn,lin,ictrl,lctrl,rctrl,cctrl,x_l0,x_u0,
     &                     kloop,.true.,xkkt,viol,iactblo,iactbhi,
     &                     iactc)

!  print a few things
 4000 do ipr=6,9,3
        if ((ipr.eq.6.and.iprint.ge.2).or.ipr.eq.9) then
          if (ni+ne.gt.0) then
            write(ipr,*)' '
            write(ipr,8900) xlamsml
            write(ipr,8901) xlambig
          endif
          write(ipr,*)' '
          write(ipr,8902) xkkt
!           if (fapriori.lt.big/10.d0) then
!             write(ipr,*)' '
!             write(ipr,8903) f-fapriori
!             if (ipr.eq.6) then
!               if (ne.ne.0) then
!                 write(47,*) mxloop,n,ni,ne
!               else
!                 write(47,*) mxloop,n,ni,ne,f-fapriori,
!      &                      xkkt,kloop,llooptot
!               endif
!             endif
!           endif
```

```
        endif
      enddo
!
      if (yhi.gt.1.d-6) then
        call open_warn_file (warn)
        write (14,8801)yhi
      endif
!
      if (iprint.ge.3) then
        do i=1,n
          write(8,5001) i,x0(i),x(i),x_l0(i),x_u0(i)
        enddo
      endif
!
      if (iprint.ge.3) then
        do j=1,ni
          write(13,8501) j,c(j),xlam(j)
          if (xlam(j).ge.biglam) then
            write(9,8600) j
            write(13,8601) j
            call open_warn_file (warn)
            write(14,8602) j,biglam
          endif
        enddo
      endif
!
      open (48,file='CUTEr.SAOi.out',position='append')
      write(48,*) n,ni,kloop,llooptot,f,viol
      close(48)

      if (debug) then
        close(12)
        open (12 ,file='Subproblems.out',status='old',position='append')
      endif
!
      return
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
 1504 format (2(1x,i6,'&'),2(1x,i6,'&'),1x,e14.7,1x,
     &        '&',e11.4,1x,'&',1x,1f8.2,1x,'&',1x,A,'\\')

 5000 format (/,20x,'SAOi algorithm: primal variables',
     &     //,100('-'),/,1x,
     &    'Component  starting point     final point  ',
     &    '   lower bound     upper bound ',/,100('-'))
 5001 format (i10,6es16.6)
!
 6000 format (/,46x,'SAOi algorithm',//,130('-'),/,1x,
     &    ' Outer  Inner   Function val  Max constr Stats     ',
     &    ' dml       Qual      xnorm       Frel      ActLo      ActHi',
     &    '       ActC  Message',/,130('-'))
 6001 format (/,26x,'SAOi algorithm',//,137('-'),/,1x,
     & '  Outer Inner SubProb        gamma         ',
     & '  skkt             f             h',
```

```
     & '   comments',/,137('-'))
 6002 format (/,9x,'SAOi algorithm: constraints and ',
     & 'dual variables',//,77('-'),/,1x,
     & 'Number                    Constraint            ',
     & '   Dual variable  ',/,77('-'))
 6010 format (/,46x,'SAOi algorithm',//,134('-'),/,1x,
     &    '  Outer Inner   Function val   Max con  S    ',
     &    '   dml      Qual       kkt     xnorm      Frel  ',
     &    '    ActLo      ActHi',
     &    '       ActC  Message',/,134('-'))
!
 6011 format (/,46x,'SAOi algorithm',//,134('-'),/,
     &        2x,'Outer',2x,'Inner',3x,'Function val',2x,
     &        'Max constr',2x,'S',8x,'dml',7x,'Qual',8x,'kkt',
     &        6x,'xnorm',5x,'Frel',4x,'ActLo',4x,'ActHi',
     &        5x,'ActC',2x,'Message',
     &        /,134('-'))
 6012 format ('  Terminated on maximum number of steps ')
 6013 format ('  Convergence criteria not satisfied - terminated on ',
     &        'maximum number of steps ')
 6014 format ('  Terminated on x-tolerance (Euclidian norm) ')
 6015 format ('  Terminated on KKT-residual ')
 6016 format ('  Terminated on x-tolerance (infinity norm) ')
 6017 format ('  Terminated on relative f-value ')
!
 7000 format (2(1x,i6),1x,es14.7,1x,es11.4,1x,l1)
 7002 format (2(1x,i6),1x,e14.7,1x,e11.4,1x,
     &        2l1,1x,e10.3,1x,e10.3,1x,e10.3,1x,e10.3,
     &        1x,i8,1x,i8,1x,i8,5x,i4,2x,A)
!
 7010 format (2(1x,i6),1x,es14.7,1x,es9.2,1x,l1)
 7012 format (2(1x,i6),1x,es14.7,1x,es9.2,1x,
     &        2l1,1x,es9.2,1x,es9.2,1x,es9.2,1x,es9.2,1x,es9.2,
     &        1x,i10,1x,i10,1x,i10,5x,i4)
!
 7500 format (/,46x,'SAOi algorithm',//,109('-'),/,1x,
     &    '  Tolerance       k       l ',/,109('-'))
 7600 format (1es12.6,2i8)
!
 8101 format (' f, viol   : ',10es18.8)
 8102 format (' x1, ...   : ',10es18.8)
 8103 format (' c1, ...   : ',10es18.8)
 8104 format (' lam1, ... : ',10es18.8)
!
 8500 format (3i7,4es14.6,3x,'-',a90)
 8501 format (1i7,2es28.12)
!
 8600 format (/,' WARNING: Dual variable ',i10,
     &          ' is on its upper bound. Severity = 10 ',33('.'),'!')
 8601 format (   ' WARNING: Dual variable ',i10,
     &          ' is on its upper bound. Severity = 10 ')
 8602 format (   ' WARNING: Dual variable ',i10,
     &          ' is on its upper bound',1es18.6,' Severity = 10 ')
!
```

```
 8800 format (/,' WARNING: relaxation variable ',i10,
     &          ' is not equal to zero, but equals: ',1es18.6,
     &          ' Severity = 7 ',33('.'),'!')
 8801 format (  ' WARNING: the relaxation variable',
     &          ' is not equal to zero, but equals: ',1es18.6,
     &          ' Severity = 7 ')
!
 8900 format (' Smallest dual variable: ',1es18.8)
 8901 format (' Largest  dual variable: ',1es18.8)
 8902 format (' KKT residual: ',1es18.8)
 8903 format (' f-f_optimal : ',1es18.8)
!
 9000 format (/,/,'  Customized output: ',1f12.2,2i7,1f8.3,10f12.4,/)
 9500 format ('    It              f              h       b&w    p-SIMP'
     &        ,'     q-GSS      brns',//,77('-'))
 9600 format (5x)
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      end subroutine sao_dense
```

E.3 Parallel global optimization algorithm

Opposed to the implementation used by Bolton *et al.* (2000) we propose a simple shared memory model that can easily run on one or more processors. This implementation is realised using the fork-join shared memory model in OpenMP.

```
      subroutine SAOi_split (n,ni,ne,x,x_l,x_u,timef,timeg,times,
     &                  ictrl,lctrl,rctrl,cctrl,u,c,cd,cm,i97,j97,
     &                  raninit,iuser,luser,cuser,ruser,nnz,eqn,lin,
     &                  mxloop,time0,ierr,inner,ini,fail)
      implicit          none
      include           'ctrl.h'
      logical           eqn(*),lin(*)
      logical           warn,feasible,raninit
      integer           i97,j97,i,n,ni,ne,it,ir,n5,kl,ll,nfe,nge,ierr
      integer           nnz,mxloop,nw,myloop
      double precision  x(n),x_l(n),x_u(n),xopt(n),f,v,fopt,vopt,p,rdm
      double precision  conv,timef,timeg,times,xkkt,xkktopt,u(97),c,cd,cm
      double precision  time0,timei,SAOi_seconds

      integer           proc,nthreads,flag,itt,inner,ifail
      integer           omp_get_num_threads,fail,ini,nfe1,nge1
      double precision  popt,pp,timegl, OMP_GET_WTIME, mytime
      common /coeff/    myloop

      external          conv,SAOi_seconds
      include           'ctrl_get.inc'

!  zero the times
      timef         = 0.d0
```

```
      timeg          = 0.d0
      times          = 0.d0

!  zero the counters
      nfe            = 0
      nge            = 0

!  initialize
      n5             = min(n,5)
      warn           = .false.
      nw             = ni+2*n

! init for parallel regeon
      fopt = 1.d16
      ir   = 0
      itt  = 0
      vopt = 0.d0
      popt = 0.d0
      p    = 0.d0
!$omp parallel default(firstprivate)
!$omp+shared(itt,u,c,cd,cm,i97,j97,raninit,p,ptarget,flag,
!$omp+fopt,xopt,xkktopt,vopt,popt,nfe,nge,nthreads,fail,ir,nfe1,nge1)
!$omp do schedule(dynamic) ordered
      do it = 1,itglobalmax

        nthreads = omp_get_num_threads()
        if (flag.ne.1) then

          write  (8,100) it
          write  (9,100) it
          write (10,100) it
          write (11,100) it
          if (debug) write (12,100) it
          write (13,100) it
          if (warn) write (14,100) it
          write (15,100) it

!  construct a random starting point
          do i=1,n
            call ranmar(rdm,1,u,c,cd,cm,i97,j97,raninit)
            x(i)=(x_u(i)-x_l(i))*(rdm)+x_l(i)
          enddo


!  call the secondary driver
          call SAOi_splita (n,ni,ne,x,x_l,x_u,timef,timeg,times,f,v,
     &                    xkkt,kl,ll,ictrl,lctrl,rctrl,cctrl,warn,nfe1,
     &                    nge1,feasible,iuser,luser,cuser,ruser,nnz,eqn,
     &                    lin,nw,u,c,cd,cm,i97,j97,raninit,mxloop,ierr)
!$omp critical (update)
          itt = itt+1
          nfe = nfe + nfe1
          nge = nge + nge1
          if(dabs(v).gt.dabs(vopt)) then
```

```
              vopt = v
            endif
!   return if there are errors
            if(ierr.lt.0) then !####
              fail = fail + 1
!             write(*,*) 'Errors were encountered in splita()'
!             stop 'Errors were encountered in splita()'
              write(*,1000) itt,ir,p,f,v,fopt,fail
!             goto 1111
!             ifail = 0
            else !####
          if (ir.eq.0) then !****
            fopt = f
            vopt = v
            do i=1,n
              xopt(i)=x(i)
            enddo
            ir = 1
            p = 1.0d0-conv(itt,ir)
            xkktopt = xkkt
            if (flag.ne.1) then
              write(*,1000) itt,ir,p,f,v,fopt,ifail
            endif
          else  !****
!         if (dabs(fopt-f).gt.tol_bayes) then
            if (dabs(fopt-f)/dabs(fopt+1.d0).gt.tol_bayes) then
              if (f.gt.fopt.or..not.feasible) then
                if (flag.ne.1) then
                  write(*,1000) itt,ir,p,f,v,fopt,ifail
                endif
              else
                fopt = f
                ir   = 1
                xkktopt = xkkt
                if (flag.ne.1) then
                  write(*,1000) itt,ir,p,f,v,fopt,ifail
                endif
              endif
            else
              if (f.lt.fopt.and.feasible) fopt=f
              ir = ir+1
              do i=1,n
                xopt(i)=x(i)
              enddo
              xkktopt = xkkt
              p  = 1.0d0-conv(itt,ir)
              if (flag.ne.1) then
                write(*,1000) itt,ir,p,f,v,fopt,ifail
              endif
!   successful search
              if (p.gt.ptarget) then
                  flag = 1
                  popt = p
                  fopt = f
```

```
            endif
          endif
!  1111      continue
        endif
        endif !****
!$omp end critical (update)

          endif !####

      enddo
!$omp end do nowait
!$omp end parallel
      p = popt
      v = vopt
      timegl = OMP_GET_WTIME() !SAOi_seconds ()
      mytime = mytime + (timegl-time0)
!       write(*,*) mytime, fail

!  successful search
      if (p.gt.ptarget) then
        write(*,2000) p
        write(*,*) ' x*       = ',(xopt(i),i=1,n5)
        write(*,*) ' '
        write(*,*) ' Nfe    = ',int(nfe/myloop),' Nge = ',int(nge/myloop)
        write(*,*) ' '
        write(*,*) ' nthreads =',nthreads, 'fail = ',fail
        write(*,*) ' '
        write(*,*) ' Time     =', mytime, 'warn = ',warn
        write(*,*) ' '
        write(*,*) ' xkktopt  =', xkktopt, 'inner',inner
        write(*,*) ' '
        write(*,*) ' vopt   =', vopt
        write(*,*) ' '
        if (fapriori.lt.big) then
          write(*,*) ' fapriori - fopt = ',fapriori-fopt,
     &             ' fopt     = ',fopt
        else
          write(*,*) ' fopt     = ',fopt
        endif
        write(*,*) ' '
        write(*,7000)
        write(*,*) ' '
!
        if (warn) then
          fail = fail + 1
!         else
!           mytime = mytime + (timegl-time0)
        endif
!         write(*,*) mytime, fail

        if (inner.eq.myloop) then

        write (*,9617) cname1,n,ni,ne,subsolver,fopt,v,
     &                 xkktopt,nfe,nge,p,ptarget,nthreads,
```

```
     &                mytime/myloop,int(fail/myloop)

189        open  (16,file='test.out',status='old',position='append')
           write (16,9615) cname1,n,ni,ne,subsolver,fopt,v,
     &                xkktopt,nfe,nge,p,ptarget,nthreads,
     &                mytime/myloop,int(fail/myloop)
           close (16)!
194        mytime = 0.d0
           endif
!            return
         else
!  unsuccessful search ..
199
           if (warn) then
             fail = fail + 1
!            else
!              mytime = mytime + (timegl-time0)
204        endif
!            write(*,*) mytime, fail

           write(*,3000)
           write(*,*) ' The best candidate for x* = ',
209  &              (xopt(i),i=1,n5)
           write(*,*) ' '
           write(*,*) ' Nfe = ',nfe,' Nge = ',nge
           write(*,*) ' nthreads =',nthreads
           write(*,*) ' '
214        if (fapriori.lt.big) then
             write(*,*) ' fapriori - fopt = ',fapriori-fopt,
     &                ' fopt = ',fopt
           else
             write(*,*) ' fopt = ',fopt
219        endif
           write(*,*) ' '
           write(*,7000)
           write(*,*) ' '
           if (inner.eq.myloop) then
224
           write (*,9617) cname1,n,ni,ne,subsolver,fopt,v,
     &                xkktopt,nfe,nge,p,ptarget,nthreads,
     &                mytime/myloop,int(fail/myloop)

229        open  (16,file='test.out',status='old',position='append')
           write (16,9615) cname1,n,ni,ne,subsolver,fopt,v,
     &                xkktopt,nfe,nge,p,ptarget,nthreads,
     &                mytime/myloop,int(fail/myloop)
           close (16)!
234        mytime = 0.d0
           endif
         endif
!
           if (warn) write (6,10)
239 !
!          open  (16,file='test.out',status='old',position='append')
```

```
!       write (16,9618) cname1,n,ni,ne,subsolver,f,v,xkktopt,
!      &                nfe,nge,p,ptarget
!       close (16)
!
      return
!
 10   format (/,' There were warnings and/or errors; see the file ',
     &        'Warnings.out',/)
 100  format (/,' Initiating global search # ',i6,5x,'<',69('='))
 1000 format (' trials = ',i6,'   successes = ',i3,
     &        '   Prob = ',1f7.4,'   f = ',1es11.4,
     &        '   h = ',1es11.4,'   fopt = ',1es14.7, '  fail = ',i6)
 2000 format (/,' Global optimum probably found; the probability is ',
     &           1f6.4,' <======= ',/)
 3000 format (/,' Global optimum possibly not found - maximum ',
     &           ' number of global iterations exceeded',/)
 6000 format (/,36x,'Global optimization using the SAOi algorithm',
     &        //,111('-'))
 7000 format (' ===> See History.out and Variables.out for ',
     &        'additional information ')
 9616 format (a24,1x,4i8,3es15.7,2i8)
 9617 format (a24,1x,4i8,3es15.7,2i8,1f9.3,' > ',1f5.3,i6,1f10.2,i6)
 9615 format (a24,1x,4i8,3es15.7,2i8,1f9.3,1f5.3,i6,1f10.2,i6)
 9618 format (a24,1x,4i8,3es15.7,2i8,1f9.3,' < ',1f5.3,' ! ')
 9626 format (4i8,3es15.7,2i8,1f10.2)
      end subroutine SAOi_split
```

E.4 BLAS-dgemv routines

Source code for interchanging optimized BLAS-dgemv routines.

E.4.1 Quadratic approximation

The first code section indicates the quadratic approximation $\tilde{f}(\boldsymbol{x})$ given in equation (7.1.1).

```
      subroutine conin_nabla2 (n,ni,x,c_a,iact,nact,x_h,bcurv,bcurvn,
     &                         c,gc,kounts)
      implicit         none
      include          'ctrl.h'
      integer          i,n,ni,kounts
      integer          nact,iact(nact)
      double precision x(n),delx(n),c_a(ni),x_h(n),c(ni),gc(ni*n),temp
      double precision bcurv(ni),bcurvn(ni*n),delx2(n)
      double precision c_a1(ni),c_a2(ni),c_a3(ni)
      double precision realtime,time1,time2,time3,time4
!
      acount = acount+1
      do i=1,n
        temp    = x(i)-x_h(i)
```

```
        delx(i) = temp
        delx2(i)= temp**2
      end do
!
!       ocl = 1
!         write(*,*) 'ocl', ocl
!
      if (ocl.eq.1) then
!  uses BLAS since Version 0.1.5
        time1 = realtime()
        call dcopy (ni,c,1,c_a,1)
        call dgemv_local ('n',ni,n,1.0d0,gc,ni,delx,1,1.d0,c_a,1)
        call dgemv_local ('n',ni,n,0.5d0,bcurvn,ni,delx2,1,1.d0,c_a,1)
        time2 = realtime()
!         write(*,1000) 1.d3*(time2-time1)
!
      elseif (ocl.eq.2) then
!  uses OpenCL since Version 0.7.6
        time3 = realtime()
        call ocl_dgemv(ocl_context_ptr,n,ni,gc,delx,c_a1,
     &                 bcurvn,delx2,c_a2,buffer,buffersize,kounts)
        do i=1,ni
          c_a(i) = c(i) + c_a1(i) + 0.5d0*c_a2(i)
        enddo
        time4 = realtime()
!         write(*,1002) 1.d3*(time4-time3)
!
      elseif (ocl.eq.3) then
        call testcublas(%val(n),%val(ni),gc,delx,c_a1,
     &                  bcurvn,delx2,c_a2,%val(kounts),cubuffer)
        do i=1,ni
          c_a(i) = c(i) + c_a1(i) + 0.5d0*c_a2(i)
        enddo
      elseif (ocl.eq.4) then
        call dcopy (ni,c,1,c_a,1)
        call dgemv ('n',ni,n,1.0d0,gc,ni,delx,1,1.d0,c_a,1)
        call dgemv ('n',ni,n,0.5d0,bcurvn,ni,delx2,1,1.d0,c_a,1)
      else
        stop 'multiplication method not defined'
      endif
!       stop
!
      return
 1000 format('Blas time = ',1f10.2,'ms')
 1002 format('OCL  time = ',1f10.2,'ms')
 1001 format('Hex value of ocl_context_ptr = ',z20)
      end subroutine conin_nabla2
```

E.4.2 Quadratic approximation

Similarly, the dual approximate subproblem from Section 2.2.3, where inverse matrix-vector multiplication is required.

```
      subroutine falk_dq (nprimal,ni,xprimal,xdual,x_l,x_u,x_h,
     &                    f,c,gf,gc,f_a,c_a,acurv,bcurv,acurvn,
     &                    bcurvn,g,kounts,iact,nact,flambda,
     &                    ictrl,lctrl,rctrl,yhi,xdual_h)
      implicit          none
      include           'ctrl.h'
      integer           i,j,k,nprimal,ni,kounts,nact,iact(nact),ilo,ihi
      integer           imeth
      double precision  xdual(ni),x_l(nprimal),x_u(nprimal),zero,one,two
      double precision  xprimal(nprimal),gf(nprimal),gc(ni,nprimal)
      double precision  acurv,bcurv(ni),acurvn (nprimal),gcji,bcji
      double precision  bcurvn(ni,nprimal),c(ni),f,f_a,c_a(ni)
      double precision  temp1,temp2,beta,flambda,g(ni),xdual_h(ni)
      double precision  x_h(nprimal),xdualj,y,sum1,yhi
      double precision  temp1n(nprimal),temp2n(nprimal)
      double precision  temp1nn(nprimal),temp2nn(nprimal)
      data              zero /0.d0/,one /1.d0/,two /2.d0/
!       data              imeth /3/
      include           'ctrl_get.inc'

!  construct the bounded dual
      if (ocl.eq.0) then
        do i=1,nprimal
          temp1 = 0.d0
          temp2 = 0.d0
          do k = 1,nact
            j = iact(k)
            xdualj=xdual(j)
            if (xdualj.gt.0.d0) then
              gcji=gc(j,i)
              if (gcji.ne.0.d0) then
                temp1 = temp1 + xdualj*gcji
              endif
              bcji=bcurvn(j,i)
              if (bcji.gt.0.d0) then
                temp2 = temp2 + xdualj*bcji
              endif
            endif
          enddo
!
          beta = x_h(i)-(gf(i) + temp1)/(acurvn(i)+temp2)
          xprimal(i) = min(x_u(i),max(x_l(i),beta))
        enddo
!
      elseif ((ocl.eq.1).or.(ocl.eq.2)) then
        call dcopy (nprimal,gf,1,temp1n,1)
        call dgemv_local ('t',ni,nprimal,1.0d0,gc,ni,xdual,1,1.d0,
     &                    temp1n,1)
        call dcopy (nprimal,acurvn,1,temp2n,1)
        call dgemv_local ('t',ni,nprimal,1.0d0,bcurvn,ni,xdual,1,1.d0,
     &               temp2n,1)
!
        do i=1,nprimal
```

```
54            beta = x_h(i)-temp1n(i)/temp2n(i)
              xprimal(i) = min(x_u(i),max(x_l(i),beta))
            enddo
          elseif (ocl.eq.3) then ! for the GPU; enforce CUBLAS !

59          call testcublas_trans(%val(nprimal),%val(ni),gc,xdual,temp1nn,
         &                  bcurvn,xdual,temp2nn,%val(kounts+1),
         &                  transcubuffer)
    !
            do i=1,nprimal
64            temp1n(i) = gf(i) + temp1nn(i)
              temp2n(i) = acurvn(i) + temp2nn(i)
            enddo
    !
            do i=1,nprimal
69            beta = x_h(i)-temp1n(i)/temp2n(i)
              xprimal(i) = min(x_u(i),max(x_l(i),beta))
            enddo
          elseif (ocl.eq.4) then ! for the GPU; enforce BLAS !
            call dcopy (nprimal,gf,1,temp1n,1)
74          call dgemv ('t',ni,nprimal,1.0d0,gc,ni,xdual,1,1.d0,temp1n,1)
            call dcopy (nprimal,acurvn,1,temp2n,1)
            call dgemv ('t',ni,nprimal,1.0d0,bcurvn,ni,xdual,1,1.d0,
         &               temp2n,1)
    !
79          do i=1,nprimal
              beta = x_h(i)-temp1n(i)/temp2n(i)
              xprimal(i) = min(x_u(i),max(x_l(i),beta))
            enddo
          else
84          stop 'ocl Undefined in Falk.f'
          endif

    !  evaluate the single relaxation variable y if needed
          if (relax) then
89          sum1 = zero
            do k = 1,nact
              j = iact(k)
              sum1=sum1+xdual(j)
            enddo
94          y = (sum1-pen1)/pen2       !   we desire biglam > pen1   !
            y = min(ymax,max(zero,y))
          else
            y = zero
          endif
99
    !  have updated xprimal; now get function value to maximize the dual
          kounts = kounts + 1
          call fun_a (nprimal,xprimal,f_a,x_h,acurv,acurvn,f,gf)
          call conin_a (nprimal,ni,xprimal,c_a,iact,nact,x_h,bcurv,
104       &              bcurvn,c,gc,kounts)
          flambda = -f_a
          do k = 1,nact
            j = iact(k)
```

```
        flambda = flambda - xdual(j)*c_a(j)
      enddo

!  compute gradient g
      do k = 1,nact
        j = iact(k)
        g(j) = -c_a(j)
      enddo

!  do some corrections for relaxation
      if (relax) then
        flambda = flambda - pen1*y - pen2*y**2/two
        do k = 1,nact
          j = iact(k)
          flambda = flambda + xdual(j)*y
          g(j) = g(j) + y
        enddo
        yhi = y
      endif
!
      return
      end
```

Appendix F

Histories

These iteration histories are for the interested reader, these show how the problem properties change during execution and how the parallel algorithm described in Chapter 5 enables the SAO program to select the most effective solver on each step in the solution trajectory. Another benefit that becomes apparent is the ability to suppress errors and to allow for certain exit conditions which is not possible during the execution of a sequential program.

F.1 Cam

These are the trajectories for the cam problem defined in Section 4.4.2. This is a well known problem, infamous because of its difficulty to solve with many solvers, especially the `l-BFGS-b` solver.

Only problem sizez of n=2000 are displayed. Similar properties are displayed for other problem sizes.

Table F.1: Parallel trajectory for cam problem, $n = 2000$

SAOi algorithm

Outer	Inner	Function val	Max constr	S	dml	Qual	kkt	xnorm	Frel	ActLo	ActHi	ActC	Message
0	0	-4.7123890E+00	7.5000E-01	F									
1	0	-0.4337959E+01	0.5078E+00	FF	0.200E+00	0.161E-01	0.676E+01	-0.655E-01	0	0	2140	5	E04NQF
2	0	-0.4562090E+01	0.1880E+00	FF	0.200E+00	0.100E+01	0.577E+01	0.420E-01	0	0	2221	5	E04NQF
3	0	-0.4725032E+01	0.4574E-03	FF	0.200E+00	0.144E+01	0.674E+01	0.293E-01	1	1	2000	5	E04NQF
4	0	-0.4645652E+01	0.1796E-03	FF	0.200E+00	0.967E+00	0.608E+01	-0.139E-01	0	4	2303	5	E04NQF
5	0	-0.4301003E+01	0.3085E-04	FF	0.200E+00	0.100E+01	0.581E+01	-0.610E-01	0	2	2476	5	E04NQF
6	0	-0.4270891E+01	0.1791E-07	TF	0.200E+00	0.100E+01	0.486E+00	-0.568E-02	0	37	2596	0	E04NQF
7	0	-0.4273425E+01	0.8535E-10	TF	0.200E+00	0.100E+01	0.478E-01	0.481E-03	0	128	2596	0	GALAHAD
8	0	-0.4273427E+01	0.5573E-12	TT	0.200E+00	0.100E+01	0.339E-04	0.309E-06	0	128	2596	0	GALAHAD

```
 Terminated on x-tolerance (Euclidian norm)
 Terminated on KKT-residual

 number of variables n    :         2000
 number of constraints ni :         6004

 Outer iterations k used  :            8
 Inner iterations l used  :            0
 Subproblem evaluations   :         1866

 Machine precision        :   2.22044604925031308E-016

f, viol   :    -4.27342703E+00    5.57331958E-13
x1, ...   :     1.00000039E+00    1.00000118E+00    1.00000237E+00    1.00000394E+00    1.00000592E+00
c1, ...   :     5.06705788E-13    5.06705788E-13    5.06705788E-13    5.06261699E-13    5.06705788E-13
lam1, ... :     3.97166734E+03    3.96803926E+03    3.96440652E+03    3.96076913E+03    3.95712712E+03

Smallest dual variable:     2.22078516E-11
Largest  dual variable:     3.97891104E+03

KKT residual:     8.52517710E-07

 Elapsed time (wall clock seconds) =     103.15
                                           0.00 (  0.00%) in evaluating f_j
                                           5.32 (  5.15%) in evaluating df_j

                                           5.32 (  5.15%) in evaluating f_j and df_j
                                          90.24 ( 87.48%) in solving the subroblems
                                           7.60 (  7.37%) in overheads
```

Table F.2: Serial trajectory for cam problem–LSQP, $n = 2000$

```
                                        SAOi algorithm

-------------------------------------------------------------------------------------------------------------------
Outer Inner   Function val   Max constr Stats     dml        Qual       xnorm        Frel ActLo ActHi  ActC  Message
-------------------------------------------------------------------------------------------------------------------
   0      0 -4.7123890E+00  7.5000E-01 F
   1      0 -4.7128601E+00  3.2855E-01 FF  2.000E-01 -1.271E-02  2.740E-01  8.247E-05     0     0  1999        -5
   2      0 -4.7126693E+00  1.6351E-01 FF  2.000E-01  9.962E-01  3.835E-01 -3.340E-05     0     0  1998        -5
   3      0 -4.7128144E+00  7.9329E-02 FF  2.000E-01  1.008E+00  4.816E-01  2.541E-05     0     0  2002        -5
   4      0 -4.7130208E+00  5.2670E-02 FF  2.000E-01  1.010E+00  6.375E-01  3.612E-05     0     0  2001        -5
   5      0 -4.7131494E+00  4.8403E-02 FF  2.000E-01  1.024E+00  7.806E-01  2.252E-05     0     0  2009        -5
   6      0 -4.7130579E+00  1.7668E-02 FF  2.000E-01  9.574E-01  9.025E-01 -1.602E-05     0     0  2011        -5
   7      0 -4.7130632E+00  8.1295E-03 FF  2.000E-01 -7.998E-01  1.552E+00  9.365E-07     0     0  2021        -5
   8      0 -4.7130593E+00  3.2272E-03 FF  2.000E-01  1.298E-01  2.284E+00 -6.808E-07     0     0  2049       -16
   9      0 -4.7130458E+00  1.0133E-03 FF  2.000E-01  2.005E-01  3.283E+00 -2.365E-06     0     0  2320       -16
  10      0 -4.7119280E+00  2.7998E-05 FF  2.000E-01  9.052E-01  4.837E+00 -1.957E-04     0     0  2393       -16
  11      0 -4.7053839E+00  1.4965E-05 FF  2.000E-01  9.762E-01  5.643E+00 -1.146E-03     0     0  1984       -16
  12      0 -4.4139707E+00  4.1144E-07 TF  2.000E-01  9.996E-01  4.985E+00 -5.108E-02     0     0  1988       -16
  13      0 -4.2204049E+00  7.9341E-13 TF  2.000E-01  9.997E-01  3.482E+00 -3.575E-02     0    24  2495         0
  14      0 -4.2729673E+00  2.9843E-13 TF  2.000E-01  1.000E+00  1.037E+00  1.007E-02     0   127  2596         0
  15      0 -4.2734272E+00  6.7990E-13 TF  2.000E-01  1.000E+00  1.019E-02  8.722E-05     0   128  2596         0
  16      0 -4.2734272E+00  6.3105E-13 TT  2.000E-01  1.000E+00  8.588E-07 -9.232E-09     0   128  2596         0

 Terminated on x-tolerance (Euclidian norm)
 Terminated on x-tolerance (infinity norm)

 number of variables n    :         2000
 number of constraints ni :         6004

 Outer iterations k used  :           16
 Inner iterations l used  :            0
 Subproblem evaluations   :         4917

 Machine precision        :   2.220446049250313O8E-016

f, viol   :    -4.27342717E+00    6.31050767E-13
x1, ...   :     1.00000039E+00    1.00000118E+00    1.00000237E+00    1.00000394E+00    1.00000592E+00
c1, ...   :     5.74207348E-13    5.74207348E-13    5.73763259E-13    5.73763259E-13    5.74207348E-13
lam1, ... :     3.97166921E+03    3.96804113E+03    3.96440839E+03    3.96077100E+03    3.95712899E+03

 Smallest dual variable:     2.54913797E-11
 Largest  dual variable:     3.97891290E+03

 KKT residual:     7.18153630E-08

 Elapsed time (wall clock seconds) =     103.89
                                           0.00 (  0.00%) in evaluating f_j
                                           3.64 (  3.51%) in evaluating df_j

                                           3.64 (  3.51%) in evaluating f_j and df_j
                                          88.69 ( 85.36%) in solving the subroblems
                                          11.56 ( 11.13%) in overheads
```

Table F.3: Serial trajectory for cam problem–`E04NQF`, $n = 2000$

SAOi algorithm

Outer	Inner	Function val	Max constr	Stats	dml	Qual	xnorm	Frel	ActLo	ActHi	ActC	Message
0	0	-4.7123890E+00	7.5000E-01	F								
1	0	-4.3379592E+00	5.0783E-01	FF	2.000E-01	1.614E-02	6.757E+00	-6.555E-02	0	0	2140	5
2	0	-4.5620903E+00	1.8802E-01	FF	2.000E-01	1.001E+00	5.771E+00	4.199E-02	0	0	2221	5
3	0	-4.7250315E+00	4.5741E-04	FF	2.000E-01	1.444E+00	6.743E+00	2.929E-02	1	1	2000	5
4	0	-4.6456525E+00	1.7958E-04	FF	2.000E-01	9.673E-01	6.078E+00	-1.387E-02	0	4	2303	5
5	0	-4.3010031E+00	3.0847E-05	FF	2.000E-01	9.995E-01	5.811E+00	-6.105E-02	0	2	2476	5
6	0	-4.2708906E+00	1.7910E-08	TF	2.000E-01	1.000E+00	4.858E-01	-5.681E-03	0	37	2596	0
7	0	-4.2734243E+00	6.3223E-08	TF	2.000E-01	1.000E+00	4.782E-02	4.807E-04	0	32	2596	0
8	0	-4.2734243E+00	2.1778E-12	TT	2.000E-01	1.000E+00	6.880E-04	5.222E-09	0	33	2595	0
9	0	-4.2734245E+00	1.1159E-08	TT	2.000E-01	1.001E+00	4.508E-04	3.644E-08	0	33	2596	0
10	0	-4.2734229E+00	2.1787E-12	TT	2.000E-01	9.997E-01	6.266E-04	-2.993E-07	0	33	2595	0
11	0	-4.2734235E+00	6.9845E-10	TT	2.000E-01	1.000E+00	5.573E-04	1.049E-07	0	34	2594	0
12	0	-4.2734255E+00	6.5923E-09	TT	2.000E-01	1.000E+00	8.353E-04	3.853E-07	0	35	2593	0
13	0	-4.2734239E+00	2.6432E-12	TT	2.000E-01	9.999E-01	7.795E-04	-3.146E-07	0	32	2596	0
14	1	-4.2734246E+00	3.2652E-10	TT	1.000E-01	1.000E+00	2.580E-04	1.376E-07	0	41	2596	0
15	0	-4.2734229E+00	1.8670E-12	TT	2.000E-01	9.991E-01	6.266E-04	-3.131E-07	0	33	2595	0
16	1	-4.2734244E+00	1.7406E-10	TT	1.000E-01	1.000E+00	5.519E-04	2.767E-07	0	41	2595	0
17	0	-4.2734235E+00	1.8670E-12	TT	2.000E-01	9.999E-01	3.335E-04	-1.718E-07	0	34	2594	0
18	0	-4.2734255E+00	6.5930E-09	TT	2.000E-01	1.001E+00	8.354E-04	3.854E-07	0	35	2593	0
19	0	-4.2734239E+00	1.0880E-12	TT	2.000E-01	9.999E-01	7.791E-04	-3.146E-07	0	32	2596	0
20	1	-4.2734246E+00	3.2652E-10	TT	1.000E-01	1.000E+00	2.580E-04	1.376E-07	0	41	2596	0
21	0	-4.2734229E+00	1.8670E-12	TT	2.000E-01	9.991E-01	6.266E-04	-3.130E-07	0	33	2595	0
22	1	-4.2734230E+00	2.1787E-12	TT	1.000E-01	1.000E+00	8.217E-06	1.034E-08	0	39	2595	0

Terminated on x-tolerance (Euclidian norm)

```
 number of variables n    :        2000
 number of constraints ni :        6004

 Outer iterations k used  :          22
 Inner iterations l used  :           4
 Subproblem evaluations   :       17146

 Machine precision        :  2.22044604925031308E-016

 f, viol   :   -4.27342300E+00   2.17870166E-12
 x1, ...   :    1.00000039E+00   1.00000118E+00    1.00000237E+00    1.00000394E+00    1.00000592E+00
 c1, ...   :   -6.66133815E-15   9.32587341E-15   -1.64313008E-14    8.43769499E-15   -4.44089210E-16
 lam1, ... :    3.97354347E+03   3.96991425E+03    3.96628036E+03    3.96264182E+03    3.95899865E+03

 Smallest dual variable:    0.00000000E+00
 Largest  dual variable:    3.98078945E+03

 KKT residual:     2.19495144E-02

 Elapsed time (wall clock seconds) =     112.47
                                           0.00 (  0.00%) in evaluating f_j
                                           4.91 (  4.36%) in evaluating df_j

                                           4.91 (  4.37%) in evaluating f_j and df_j
                                          91.07 ( 80.97%) in solving the subroblems
                                          16.49 ( 14.67%) in overheads
```

F.2 Modified *n*-variate cantilever beam

These are the trajectories for the Modified *n*-variate cantilever beam problem defined in Section 4.4.4. Only problem sizez of n=2000 are displayed. Similar properties are displayed for other problem sizes.

Table F.4: Parallel trajectory for n-variate cantilever beam problem, $n = 2000$

```
                                    SAOi algorithm

-------------------------------------------------------------------------------------------------------------------------------
  Outer  Inner   Function val  Max constr  S        dml        Qual        kkt       xnorm      Frel     ActLo     ActHi      ActC  Message
-------------------------------------------------------------------------------------------------------------------------------
      0      0  1.5600000E+00  2.2204E-16 T
      1      0  0.1316960E+01  0.1110E+00 FF  0.200E+00  0.171E+01  0.570E+02  0.949E-01         0         0         1         0  Dual
      2      0  0.1306694E+01  0.2392E-01 FF  0.200E+00 -0.665E+00  0.229E+02  0.443E-02         0         0         1         0  Dual
      3      0  0.1309841E+01  0.2990E-02 FF  0.200E+00  0.485E+00  0.594E+01 -0.136E-02         0         0         1         0  Dual
      4      0  0.1310269E+01  0.3507E-03 FF  0.200E+00  0.511E+00  0.144E+01 -0.185E-03         0         0         1         0  Dual
      5      0  0.1310313E+01  0.4382E-04 FF  0.200E+00  0.467E+00  0.355E+00 -0.193E-04         0         0         1         0  Dual
      6      0  0.1310320E+01  0.6528E-05 TF  0.200E+00  0.530E+00  0.848E-01 -0.284E-05         0         0         1         0  Dual
      7      0  0.1310322E+01  0.9722E-07 TF  0.200E+00  0.984E+00  0.571E-02 -0.118E-05         0         0         1         0  Dual

 Terminated on KKT-residual

 number of variables n    :         2000
 number of constraints ni :            1

 Outer iterations k used  :            7
 Inner iterations l used  :            0
 Subproblem evaluations   :        15356

 Machine precision        :   2.220446049250313O8E-016

f, viol   :     1.31032232E+00    9.72208238E-08
x1, ...   :     6.29882304E+00    6.29724775E+00    6.29567206E+00    6.29409597E+00    6.29251949E+00
c1, ...   :     9.72208238E-08
lam1, ... :     4.36774090E-01

Smallest dual variable:     4.36774090E-01
Largest  dual variable:     4.36774090E-01

KKT residual:     6.75981655E-07

 Elapsed time (wall clock seconds) =      52.14
                                           0.00 (  0.00%) in evaluating f_j
                                           0.00 (  0.00%) in evaluating df_j

                                           0.00 (  0.01%) in evaluating f_j and df_j
                                          52.08 ( 99.89%) in solving the subroblems
                                           0.05 (  0.10%) in overheads
```

Table F.5: Serial trajectory for n-variate cantilever beam problem–LSQP, $n = 2000$

SAOi algorithm

Outer	Inner	Function val	Max constr	Stats	dml	Qual	xnorm	Frel	ActLo	ActHi	ActC	Message
0		-4.7123890E+00	7.5000E-01	F								
1	0	-4.7126735E+00	3.3356E-01	FF	2.000E-01	-7.457E-03	2.772E-01	4.980E-05	0	0	1994	-5
2	0	-4.7124928E+00	1.6474E-01	FF	2.000E-01	9.961E-01	3.778E-01	-3.163E-05	0	0	1996	-5
3	0	-4.7128092E+00	7.3915E-02	FF	2.000E-01	1.003E+00	4.557E-01	5.540E-05	0	0	1995	0
4	0	-4.7128908E+00	4.2218E-02	FF	2.000E-01	1.022E+00	5.890E-01	1.427E-05	0	0	2002	0
5	0	-4.7129392E+00	2.4210E-02	FF	2.000E-01	1.063E+00	7.581E-01	8.480E-06	0	0	2005	0
6	0	-4.7133320E+00	1.4126E-02	FF	2.000E-01	1.011E+00	9.438E-01	6.876E-05	0	0	2005	0
7	0	-4.7163271E+00	8.0611E-03	FF	2.000E-01	1.003E+00	1.227E+00	5.242E-04	0	0	2003	0
8	0	-4.7394077E+00	4.4751E-03	FF	2.000E-01	1.001E+00	1.613E+00	4.038E-03	0	0	2004	0
9	0	-4.9298049E+00	2.4356E-03	FF	2.000E-01	1.000E+00	3.580E+00	3.317E-02	0	0	2002	0
10	0	-5.4004131E+00	1.4145E-03	FF	2.000E-01	1.001E+00	8.215E+00	7.936E-02	0	26	2191	0
11	0	-5.7553318E+00	9.9824E-04	FF	2.000E-01	1.001E+00	7.753E+00	5.545E-02	0	116	2624	0
12	0	-5.6652857E+00	3.3210E-04	FF	2.000E-01	9.994E-01	3.270E+00	-1.333E-02	0	1214	2784	0
13	0	-5.5085653E+00	8.2772E-05	FF	2.000E-01	9.996E-01	3.649E+00	-2.351E-02	0	1019	2969	0
14	0	-5.2433959E+00	4.1467E-06	TF	2.000E-01	9.995E-01	5.195E+00	-4.074E-02	0	806	2915	0
15	0	-5.1900931E+00	2.1626E-06	TF	2.000E-01	9.999E-01	1.046E+00	-8.537E-03	0	770	2911	0
16	0	-5.1827175E+00	2.0711E-06	TF	2.000E-01	1.000E+00	1.437E-01	-1.192E-03	0	765	2911	0
17	0	-5.1826810E+00	2.0705E-06	TT	2.000E-01	1.000E+00	7.063E-04	-5.903E-06	0	765	2911	0
18	0	-5.1826810E+00	2.0705E-06	TT	2.000E-01	1.000E+00	4.544E-08	-4.099E-10	0	765	2911	0

Terminated on x-tolerance (Euclidian norm)
Terminated on x-tolerance (infinity norm)

```
 number of variables n     :        2000
 number of constraints ni  :        6004

 Outer iterations k used   :          18
 Inner iterations l used   :           0
 Subproblem evaluations    :        1691

 Machine precision         :   2.22044604925031308E-016

 f, viol   :   -5.18268097E+00   2.07047792E-06
 x1, ...   :    1.00000246E+00   1.00000739E+00    1.00001479E+00    1.00002465E+00    1.00003697E+00
 c1, ...   :    2.07047792E-06   2.07047792E-06    2.07047792E-06    2.07047791E-06    2.07047791E-06
 lam1, ... :    6.34217048E+02   6.32436346E+02    6.30648453E+02    6.28853421E+02    6.27051300E+02

 Smallest dual variable:     1.54595492E-10
 Largest  dual variable:     6.37758243E+02

 KKT residual:     1.32046436E-03

 Elapsed time (wall clock seconds) =      51.20
                                           0.00 (  0.00%) in evaluating f_j
                                           4.06 (  7.92%) in evaluating df_j

                                           4.06 (  7.92%) in evaluating f_j and df_j
                                          34.13 ( 66.65%) in solving the subproblems
                                          13.02 ( 25.42%) in overheads
```

Table F.6: Serial trajectory for n-variate cantilever beam problem–`E04NQF`, $n = 2000$

```
                                    SAOi algorithm

-------------------------------------------------------------------------------------------------------------------------------
   Outer Inner    Function val  Max constr Stats     dml       Qual      xnorm       Frel       ActLo      ActHi       ActC  Message
-------------------------------------------------------------------------------------------------------------------------------
      0     0  1.5600000E+00  2.2204E-16 T
      1     0  1.2637357E+00  3.4037E-01 FF  2.000E-01  1.831E+00  6.564E+01  1.157E-01          0          0          1        0
      2     0  1.2646104E+00  1.3255E-01 FF  2.000E-01  1.379E-02  3.037E+01 -3.864E-04          0          0          1        0
      3     0  1.3049252E+00  1.4620E-02 FF  2.000E-01  8.345E-01  9.721E+00 -1.780E-02          0          0          1        0
      4     0  1.3101430E+00  6.7667E-04 FF  2.000E-01  9.000E-01  1.802E+00 -2.264E-03          0          0          1        0
      5     0  1.3103046E+00  6.9365E-05 FF  2.000E-01  7.059E-01  4.004E-01 -6.995E-05          0          0          1        0
      6     0  1.3103186E+00  1.0498E-05 FF  2.000E-01  6.257E-01  1.228E-01 -6.082E-06          0          0          1        0
      7     0  1.3103224E+00  9.1777E-07 TF  2.000E-01  9.021E-01  6.376E-02 -1.641E-06          0          0          1        0
      8     0  1.3103228E+00  8.7179E-09 TF  2.000E-01  9.904E-01  8.265E-03 -1.708E-07          0          0          1        0
      9     0  1.3103228E+00  6.6835E-14 TT  2.000E-01  1.000E+00  2.449E-05 -1.654E-09          0          0          1        0

 Terminated on x-tolerance (Euclidian norm)

 number of variables n    :         2000
 number of constraints ni :            1

 Outer iterations k used  :            9
 Inner iterations l used  :            0
 Subproblem evaluations   :        16248

 Machine precision        :   2.22044604925031308E-016

f, viol   :     1.31032284E+00    6.68354261E-14
x1, ...   :     6.30427995E+00    6.29554176E+00    6.29396647E+00    6.29239078E+00    6.29081470E+00
c1, ...   :     6.68354261E-14
lam1, ... :     4.38289633E-01

Smallest dual variable:     4.38289633E-01
Largest  dual variable:     4.38289633E-01

KKT residual:     2.66376246E-05

 Elapsed time (wall clock seconds) =      55.45
                                           0.00 (  0.00%) in evaluating f_j
                                           0.00 (  0.00%) in evaluating df_j

                                           0.00 (  0.01%) in evaluating f_j and df_j
                                          55.43 ( 99.96%) in solving the subroblems
                                           0.02 (  0.03%) in overheads
```

Table F.7: Serial trajectory for n-variate cantilever beam problem–l-BFGS-b, $n = 2000$

SAOi algorithm

Outer	Inner	Function val	Max constr	Stats	dml	Qual	xnorm	Frel	ActLo	ActHi	ActC	Message
0	0	1.5600000E+00	2.2204E-16	T								
1	0	1.3187360E+00	1.0590E-01	FF	2.000E-01	1.721E+00	5.691E+01	9.424E-02	0	0	1	0
2	0	1.3076584E+00	2.2806E-02	FF	2.000E-01	-7.944E-01	2.256E+01	4.777E-03	0	0	1	0
3	0	1.3099686E+00	2.9853E-03	FF	2.000E-01	3.965E-01	6.145E+00	-1.001E-03	0	0	1	0
4	0	1.3102737E+00	3.8701E-04	FF	2.000E-01	3.953E-01	1.569E+00	-1.320E-04	0	0	1	0
5	0	1.3103125E+00	5.1558E-05	FF	2.000E-01	3.896E-01	4.007E-01	-1.682E-05	0	0	1	0
6	0	1.3103193E+00	7.5766E-06	TF	2.000E-01	4.795E-01	9.848E-02	-2.918E-06	0	0	1	0
7	0	1.3103222E+00	2.6527E-07	TF	2.000E-01	9.604E-01	9.646E-03	-1.287E-06	0	0	1	0
8	0	1.3103224E+00	2.9026E-10	TT	2.000E-01	9.989E-01	2.945E-04	-4.999E-08	0	0	1	0
9	0	1.3103224E+00	-1.0340E-10	TT	2.000E-01	1.000E+00	4.089E-07	-7.443E-11	0	0	1	0

```
 Terminated on x-tolerance (Euclidian norm)
 Terminated on x-tolerance (infinity norm)

 number of variables n    :         2000
 number of constraints ni :            1

 Outer iterations k used  :            9
 Inner iterations l used  :            0
 Subproblem evaluations   :           44

 Machine precision        :   2.22044604925031308E-016

f, viol   :     1.31032236E+00   -1.03403952E-10
x1, ...   :     6.29882316E+00    6.29724786E+00    6.29567217E+00    6.29409608E+00    6.29251960E+00
c1, ...   :    -1.03403952E-10
lam1, ... :     4.36774121E-01

Smallest dual variable:     4.36774121E-01
Largest  dual variable:     4.36774121E-01

KKT residual:     4.51641703E-11

 Elapsed time (wall clock seconds) =       0.03
                                           0.00 (  6.67%) in evaluating f_j
                                           0.00 (  3.33%) in evaluating df_j

                                           0.00 ( 10.00%) in evaluating f_j and df_j
                                           0.01 ( 33.33%) in solving the subroblems
                                           0.02 ( 56.67%) in overheads
```

F.3 Vanderplaats' cantilever

These are the trajectories for the Vanderplaats' cantilever beam problem defined in Section 4.4.3. Only problem sizez of n=2000 are displayed. Similar properties are displayed for other problem sizes.

Table F.8: Parallel trajectory for Vanderplaats' cantilever beam problem, $n = 2000$

SAOi algorithm

Outer	Inner	Function val	Max constr	S	dml	Qual	kkt	xnorm	Frel	ActLo	ActHi	ActC	Message
0	0	1.5000000E+05	-4.0476E-01	T									
1	0	0.5910025E+05	0.9355E+00	FF	0.200E+00	0.140E+01	0.473E+03	0.606E+00	0	0	525	0	GALAHAD
2	0	0.5404018E+05	0.8151E+00	FF	0.200E+00	0.308E+01	0.320E+03	0.856E-01	0	0	1582	0	GALAHAD
3	0	0.5908101E+05	0.1761E+00	FF	0.200E+00	0.872E+00	0.174E+03	-0.933E-01	37	0	1039	0	GALAHAD
4	0	0.6333578E+05	0.1608E-01	FF	0.200E+00	0.960E+00	0.792E+02	-0.720E-01	30	0	6	0	DUAL
5	0	0.6365954E+05	0.3002E-03	FF	0.200E+00	0.976E+00	0.143E+02	-0.511E-02	30	0	5	0	DUAL
6	0	0.6366509E+05	0.1573E-04	FF	0.200E+00	0.979E+00	0.184E+01	-0.871E-04	30	0	223	0	DUAL
7	0	0.6366525E+05	0.2143E-04	FF	0.200E+00	0.982E+00	0.351E+00	-0.251E-05	30	0	872	0	DUAL
8	0	0.6366516E+05	0.3053E-05	TF	0.200E+00	0.100E+01	0.708E-01	0.138E-05	30	0	977	0	DUAL
9	0	0.6366518E+05	0.2429E-05	TF	0.200E+00	0.100E+01	0.142E-01	-0.251E-06	30	0	977	0	DUAL
10	0	0.6366518E+05	0.2531E-05	TF	0.200E+00	0.100E+01	0.283E-02	0.512E-07	30	0	977	0	DUAL
11	0	0.6366518E+05	0.2511E-05	TT	0.200E+00	0.100E+01	0.567E-03	-0.101E-07	30	0	977	0	DUAL
12	0	0.6366518E+05	0.2515E-05	TT	0.200E+00	0.100E+01	0.113E-03	0.203E-08	30	0	977	0	DUAL
13	0	0.6366518E+05	0.2514E-05	TT	0.200E+00	0.100E+01	0.223E-04	-0.408E-09	30	0	977	0	DUAL

```
 Terminated on x-tolerance (Euclidian norm)

 number of variables n    :         2000
 number of constraints ni :         2001

 Outer iterations k used  :           13
 Inner iterations l used  :            0
 Subproblem evaluations   :        49741

 Machine precision        :   2.22044604925031308E-016

f, viol   :     6.36651760E+04    2.51393399E-06
x1, ...   :     3.25590953E+00    6.51181906E+01    3.25482332E+00    6.50964665E+01    3.25373639E+00
c1, ...   :    -2.23956181E-01    6.16020657E-09   -2.23955792E-01    6.09300344E-09   -2.23955403E-01
lam1, ... :     0.00000000E+00    8.13977377E-01    0.00000000E+00    8.13705825E-01    0.00000000E+00

Smallest dual variable:     0.00000000E+00
Largest  dual variable:     3.18007290E+04

KKT residual:     7.99449334E-02

 Elapsed time (wall clock seconds) =      61.74
                                           0.00 (  0.00%) in evaluating f_j
                                           0.84 (  1.37%) in evaluating df_j

                                           0.85 (  1.37%) in evaluating f_j and df_j
                                          54.85 ( 88.85%) in solving the subroblems
                                           6.04 (  9.78%) in overheads
```

Table F.9: Serial trajectory for Vanderplaats' cantilever beam problem–LSQP, $n = 2000$

SAOi algorithm

Outer	Inner	Function val	Max constr	Stats	dml	Qual	xnorm	Frel	ActLo	ActHi	ActC	Message
0	0	1.5000000E+05	-4.0476E-01	T								
1	0	5.9100247E+04	9.3548E-01	FF	2.000E-01	1.401E+00	4.727E+02	6.060E-01	0	0	525	0
2	0	5.4040181E+04	8.1514E-01	FF	2.000E-01	3.075E+00	3.203E+02	8.562E-02	0	0	1582	0
3	0	5.9081010E+04	1.7607E-01	FF	2.000E-01	8.718E-01	1.744E+02	-9.328E-02	37	0	1039	0
4	0	6.2727371E+04	3.1506E-02	FF	2.000E-01	9.536E-01	8.396E+01	-6.172E-02	30	0	976	0
5	0	6.3601237E+04	2.0805E-03	FF	2.000E-01	9.830E-01	2.343E+01	-1.393E-02	30	0	977	0
6	0	6.3662794E+04	8.0079E-05	FF	2.000E-01	9.917E-01	4.507E+00	-9.678E-04	30	0	977	0
7	0	6.3665158E+04	3.1930E-06	TF	2.000E-01	9.915E-01	9.024E-01	-3.713E-05	30	0	977	0
8	0	6.3665252E+04	1.2769E-07	TF	2.000E-01	9.915E-01	1.805E-01	-1.480E-06	30	0	977	0
9	0	6.3665256E+04	5.1091E-09	TF	2.000E-01	9.915E-01	3.610E-02	-5.919E-08	30	0	977	0
10	0	6.3665256E+04	2.0442E-10	TF	2.000E-01	9.915E-01	7.219E-03	-2.368E-09	30	0	977	0
11	0	6.3665256E+04	8.2747E-12	TF	2.000E-01	9.914E-01	1.444E-03	-9.470E-11	30	0	977	0
12	0	6.3665256E+04	4.6496E-13	TT	2.000E-01	9.915E-01	2.888E-04	-3.771E-12	30	0	977	0
13	0	6.3665256E+04	2.6734E-13	TT	2.000E-01	1.005E+00	5.775E-05	-9.463E-14	30	0	977	0

```
 Terminated on x-tolerance (Euclidian norm)

 number of variables n    :         2000
 number of constraints ni :         2001

 Outer iterations k used  :           13
 Inner iterations l used  :            0
 Subproblem evaluations   :          332

 Machine precision        :   2.22044604925031308E-016

f, viol   :     6.36652559E+04    2.67341704E-13
x1, ...   :     3.25591159E+00    6.51182318E+01    3.25482538E+00    6.50965076E+01    3.25373845E+00
c1, ...   :    -2.23957653E-01    2.13162821E-13   -2.23957265E-01    2.27373675E-13   -2.23956875E-01
lam1, ... :     1.72259002E-13    8.13977928E-01    1.72259301E-13    8.13706376E-01    1.72259601E-13

Smallest dual variable:     2.57467980E-15
Largest  dual variable:     3.18008490E+04

KKT residual:     1.48642930E-05

 Elapsed time (wall clock seconds) =       6.97
                                           0.00 (  0.03%) in evaluating f_j
                                           0.29 (  4.11%) in evaluating df_j

                                           0.29 (  4.14%) in evaluating f_j and df_j
                                           4.65 ( 66.71%) in solving the subroblems
                                           2.03 ( 29.15%) in overheads
```

Table F.10: Serial trajectory for Vanderplaats' cantilever beam problem–`E04NQF`, $n = 2000$

```
                                        SAOi algorithm

-------------------------------------------------------------------------------------------------------------------------------
   Outer Inner   Function val  Max constr Stats     dml       Qual      xnorm       Frel       ActLo      ActHi       ActC  Message
-------------------------------------------------------------------------------------------------------------------------------
      0     0  1.5000000E+05 -4.0476E-01 T
      1     0  6.7466250E+04  8.5303E-01 FF  2.000E-01  1.499E+00  4.281E+02  5.502E-01          0          0        955        10
      2     0  5.3616052E+04  8.0952E-01 FF  2.000E-01  1.556E+00  3.295E+02  2.053E-01          0          0       1648         0
      3     0  5.7065740E+04  2.6931E-01 FF  2.000E-01  4.375E-01  2.016E+02 -6.434E-02         38          0       1045         0
      4     0  6.2328640E+04  4.4569E-02 FF  2.000E-01  8.599E-01  9.705E+01 -9.222E-02         31          0        976         0
      5     0  6.3588353E+04  2.4908E-03 FF  2.000E-01  9.545E-01  2.522E+01 -2.021E-02         30          0        977         0
      6     0  6.3662628E+04  8.5427E-05 FF  2.000E-01  9.716E-01  4.663E+00 -1.168E-03         30          0        977         0
      7     0  6.3665153E+04  3.3373E-06 TF  2.000E-01  9.675E-01  9.223E-01 -3.967E-05         30          0        977         0
      8     0  6.3665252E+04  1.3405E-07 TF  2.000E-01  9.665E-01  1.849E-01 -1.546E-06         30          0        977         0
      9     0  6.3665256E+04  5.3588E-09 TF  2.000E-01  9.667E-01  3.697E-02 -6.214E-08         30          0        977         0
     10     0  6.3665256E+04  2.1437E-10 TF  2.000E-01  9.667E-01  7.394E-03 -2.484E-09         30          0        977         0
     11     0  6.3665256E+04  8.5612E-12 TF  2.000E-01  9.667E-01  1.478E-03 -9.938E-11         30          0        977         0
     12     0  6.3665256E+04  3.3529E-13 TT  2.000E-01  9.675E-01  2.926E-04 -3.975E-12         30          0        977         0
     13     0  6.3665256E+04  6.3949E-14 TT  2.000E-01  9.993E-01  3.286E-09 -1.678E-13         30          0        977         0

 Terminated on x-tolerance (Euclidian norm)
 Terminated on x-tolerance (infinity norm)

 number of variables n    :         2000
 number of constraints ni :         2001

 Outer iterations k used  :           13
 Inner iterations l used  :            0
 Subproblem evaluations   :        41078

 Machine precision        :   2.22044604925031308E-016

f, viol   :     6.36652559E+04    6.39488462E-14
x1, ...   :     3.25591147E+00    6.51182294E+01    3.25482524E+00    6.50965049E+01    3.25373831E+00
c1, ...   :    -2.23957565E-01    0.00000000E+00   -2.23957165E-01   -1.42108547E-14   -2.23956775E-01
lam1, ... :     0.00000000E+00    8.13977867E-01    0.00000000E+00    8.13706287E-01    0.00000000E+00

Smallest dual variable:     0.00000000E+00
Largest  dual variable:     3.18008423E+04

KKT residual:     3.30148056E-05

 Elapsed time (wall clock seconds) =      31.83
                                           0.00 (  0.01%) in evaluating f_j
                                           0.29 (  0.91%) in evaluating df_j

                                           0.29 (  0.92%) in evaluating f_j and df_j
                                          29.49 ( 92.66%) in solving the subroblems
                                           2.04 (  6.42%) in overheads
```

Table F.11: Serial trajectory for Vanderplaats' cantilever beam problem–`l-BFGS-b`, $n = 2000$

SAOi algorithm

Outer	Inner	Function val	Max constr	Stats	dml	Qual	xnorm	Frel	ActLo	ActHi	ActC	Message
0	0	1.5000000E+05	-4.0476E-01	T								
1	0	7.3884211E+04	3.8907E-01	FF	2.000E-01	1.257E+00	4.387E+02	5.074E-01	0	0	409	0
2	0	6.4523811E+04	1.2581E-01	FF	2.000E-01	2.125E+00	3.274E+02	1.267E-01	0	0	156	0
3	0	6.3539741E+04	4.2845E-02	FF	2.000E-01	2.186E+01	1.522E+02	1.525E-02	0	0	36	0
4	0	6.3674420E+04	2.2567E-03	FF	2.000E-01	5.719E-01	4.441E+01	-2.120E-03	19	0	1	0
5	0	6.3665667E+04	1.4642E-04	FF	2.000E-01	-1.529E+00	1.209E+01	1.375E-04	28	0	354	0
6	0	6.3665111E+04	6.9796E-06	TF	2.000E-01	1.774E+00	2.589E+00	8.731E-06	30	0	570	0
7	0	6.3665250E+04	2.8833E-07	TF	2.000E-01	9.774E-01	3.510E-01	-2.191E-06	30	0	876	0
8	0	6.3665210E+04	1.4480E-06	TF	2.000E-01	1.003E+00	6.910E-02	6.341E-07	30	0	977	0
9	0	6.3665216E+04	1.2448E-06	TF	2.000E-01	9.992E-01	1.386E-02	-9.938E-08	30	0	977	0
10	0	6.3665215E+04	1.2865E-06	TF	2.000E-01	1.000E+00	2.787E-03	2.089E-08	30	0	977	0
11	0	6.3665215E+04	1.2782E-06	TT	2.000E-01	1.000E+00	5.593E-04	-4.167E-09	30	0	977	0
12	0	6.3665215E+04	1.2798E-06	TT	2.000E-01	1.000E+00	1.119E-04	8.317E-10	30	0	977	0
13	0	6.3665215E+04	1.2795E-06	TT	2.000E-01	1.000E+00	2.240E-05	-1.697E-10	30	0	977	0

```
 Terminated on x-tolerance (Euclidian norm)

 number of variables n    :         2000
 number of constraints ni :         2001

 Outer iterations k used  :           13
 Inner iterations l used  :            0
 Subproblem evaluations   :         4932

 Machine precision        :   2.22044604925031308E-016

f, viol   :     6.36652152E+04    1.27947816E-06
x1, ...   :     3.25591054E+00    6.51182108E+01    3.25482433E+00    6.50964866E+01    3.25373740E+00
c1, ...   :    -2.23956901E-01    1.05117692E-10   -2.23956512E-01    1.06027187E-10   -2.23956123E-01
lam1, ... :     0.00000000E+00    8.13977634E-01    0.00000000E+00    8.13706082E-01    0.00000000E+00

Smallest dual variable:     0.00000000E+00
Largest  dual variable:     3.18007879E+04

KKT residual:     4.06884137E-02

 Elapsed time (wall clock seconds) =       9.96
                                           0.00 (  0.00%) in evaluating f_j
                                           0.29 (  2.89%) in evaluating df_j

                                           0.29 (  2.89%) in evaluating f_j and df_j
                                           7.63 ( 76.64%) in solving the subroblems
                                           2.04 ( 20.47%) in overheads
```